DISEÑO Y DESARROLLO DE UNA PLATAFORMA DE PARALELIZACIÓN Y DISTRIBUCIÓN AUTOMÁTICA PARA SUPERCOMPUTACIÓN NO COORDINADA

DESIGN AND DEVELOPMENT OF A PARALLELIZATION AND AUTOMATIC DISTRIBUTION PLATFORM FOR UNCOORDINATED SUPERCOMPUTATION

JUAN RAMOS DÍAZ

MÁSTER EN INVESTIGACIÓN EN INFORMÁTICA, FACULTAD DE INFORMÁTICA,
UNIVERSIDAD COMPLUTENSE DE MADRID

Trabajo Fin Máster en Ingeniería Informática

05 de Julio de 2019

Director.:Juan Pavón Mestras
Colaborador: Jose Javier Garcia Aranda (NOKIA)

Convocatoria: Junio-Julio 2019
Calificación: Sobresaliente - 10

Autorización de Difusión

JUAN RAMOS DÍAZ

17/06/2019

El abajo firmante, matriculado en el Máster en Investigación en Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Máster: "DISEÑO Y DESARROLLO DE UNA PLATAFORMA DE PARALELIZACIÓN Y DISTRIBUCIÓN AUTOMÁTICA PARA SUPERCOMPUTACIÓN NO COORDINADA", realizado durante el curso académico 2018-2019 bajo la dirección de Juan Pavón Mestras y con la colaboración externa de dirección de Jose Javier García Aranda (NOKIA), en el Departamento de Ingeniería del Software e Inteligencia Artificial, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Resumen

En esta memoria se presenta el diseño y desarrollo de una plataforma de paralelización y distribución de código automática destinada a la supercomputación de manera no coordinada.

El desarrollo de esta aplicación se enmarca en el proyecto de I+D+i, CLOUDBOOK: A Cloud-aware distributed parallel compiler (Celtic plus C2016/2-4), cuyo objetivo es la creación de una plataforma que integre la distribución a través de la nube y la compilación y ejecución de un programa en una única plataforma, e integrar distintas tecnologías de supercomputación en esta plataforma.

Dentro de este proyecto surge la necesidad de desarrollar la plataforma encargada de realizar el troceado y distribución automática de código de forma transparente al programador. De estos dos problemas, el de mayor contribución al estado del arte es el del troceado automático del código para su ejecución de forma distribuida y paralela.

Esta será la plataforma desarrollada en este trabajo, cuyo objetivo es la automatización de la generación de piezas de código distribuible a partir del programa fuente original, así como la distribución de dichas piezas en una red de máquinas. Esto permite liberar al programador de realizar un diseño distribuido, sin renunciar a las ventajas de la computación distribuida con un coste de aprendizaje de uso muy bajo.

Con este objetivo se ha desarrollado la plataforma en el lenguaje de programación Python, usando librerías para analizar el lenguaje y su estructura sintáctica, como PLY. Para la comunicación a través de la red se ha usado el framework Flask de Python.

Se ha usado también NFS como sistema de ficheros distribuido sobre el que están los ficheros del programa. Por último, la información de configuración se realiza mediante ficheros en formato JSON

Palabras clave

Supercomputación, computación paralela, computación distribuida, compilación, Python, Flask, NFS, JSON, PLY.

Summary

This work presents the design and development of parallelization and automatic code distribution platform destined for supercomputing in an uncoordinated manner.

This application has been developed in research project CLOUDBOOK: A Cloud-aware distributed parallel compiler (Celtic plus C2016/2-4), whose objective is the creation of a platform that allows the parallelization of programs to be distributed in the cloud , and facilitate the integration with different supercomputing technologies.

A fundamental requirement for this project is the need for making the automatic distribution of code transparently to the programmer. The main contribution to the state of the art is the automatic division of the code for its execution in a distributed and parallel manner.

This will be the platform developed in this work, whose objective is the automation of the generation of distributable code pieces from the original source program, as well as the distribution of said parts in a network of machines. This allows the programmer to make a distributed design without giving up the advantages of distributed computing with a very low use learning cost.

With this objective, the platform has been developed in the Python programming language, using libraries to analyze the language and its syntactic structure, such as PLY. For communication through the network, the Python framework, Flask, has been used.

NFS has also been used as a distributed file system on which the program files are located. Finally, the configuration information is done through files in JSON format.

Keywords

Supercomputing, parallel computing, distributed computing, compilation, Python, Flask, NFS, JSON, PLY.

Índice de contenidos

Índice de figuras

Índice de tablas

Agradecimientos

A mi familia que me ha apoyado en todo este trayecto.

A Jose Javier por su guía y ayuda incansable a lo largo de este proyecto.

A Juan por su ayuda y buenos consejos a lo largo del proyecto.

A Celia por su ayuda, apoyo constante y ánimos incondicionales.

1. Introducción, objetivos y estructura del documento

En este capítulo se introduce el proyecto presentado en este trabajo, explicando los objetivos que se quieren cumplir, también se explica la metodología de trabajo seguida, y por último se explica la estructura de esta memoria.

1.1. Introducción

A medida que los procesadores son más rápidos y efectivos han aparecido restricciones físicas que impiden que la frecuencia de los mismos siga escalando (1). A su vez, la reducción del consumo energético ha cobrado importancia en los últimos años (2) debido a esto la programación paralela se ha convertido en el paradigma a seguir en la computación, tanto a nivel de multiprocesadores en los ordenadores domésticos (2) como para la supercomputación. Al igual que ha pasado con la computación distribuida, necesaria para gestionar redes de telecomunicaciones, aplicaciones en red, protocolos de acceso en tiempo real y la propia computación paralela (3).

Estas necesidades computacionales han dado lugar a muchas tecnologías que para mejorar la eficiencia y maximizar los recursos y reducir los costes y consumo explotan el paralelismo: OpenMP, CUDA en tarjetas gráficas; la distribución: BOINC, Charm++; o ambas: Apache Spark y Apache Flink en el campo de análisis de Big Data.

Es por esto por lo que paradigmas de computación que mezclen computación paralela y distribuida son cada vez más necesarios, en este contexto surge la computación grid, clúster, y computación en la nube.

La computación grid es un tipo de computación distribuida en la que se compone una supercomputadora virtual mediante la unión débilmente acoplada de muchas computadoras trabajando juntas para realizar una tarea compleja, este trabajo se puede realizar de forma simultánea o secuencial (4).

Estas tecnologías se enfrentan a dos problemas principales, el diseño del código para que se pueda distribuir o paralelizar, y la distribución y ejecución del código previamente preparado en distintas arquitecturas, o máquinas en distintas redes.

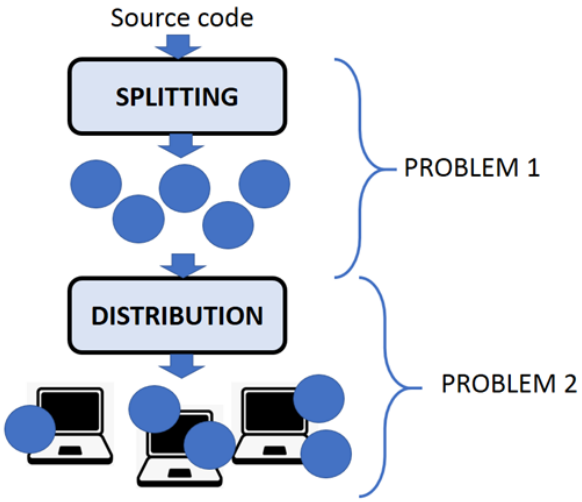


Figura 1: Problemas que se afrontan

El primer problema se puede abordar de dos maneras, ya sea creando código especialmente preparado para su futura distribución y ejecución en paralelo, o realizando la transformación del código de forma automática. En el primer caso se le exige mucho conocimiento al programador ya que los algoritmos paralelos son más difíciles de escribir (5), y el segundo caso requiere un control de grano muy fino sobre el código fuente y el compilador.

En este trabajo se ofrece otra posibilidad de aprovechar la distribución y paralelismo sin exigir al programador un cambio de paradigma en el lenguaje de programación o de realizar un diseño del programa distribuido.

Para ello se va a diseñar y desarrollar una plataforma que, usando como unidad mínima de proceso las funciones del programa en lugar de los bucles o los datos, realice un proceso de *trans-compilación* que traduce el código fuente de un programa en código que permita a la plataforma distribuirlo y ejecutarlo de forma transparente, siguiendo el modelo de la computación grid, para ofrecer un supercomputador formado por todas las computadoras distribuidas (véase la Figura 2).

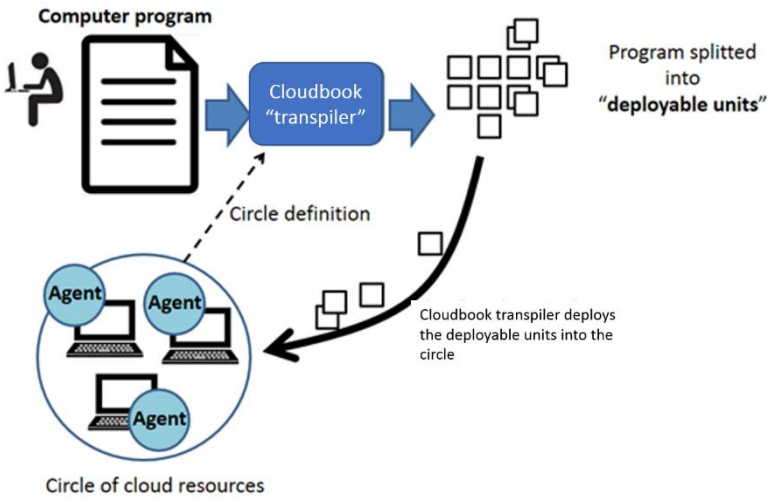


Figura 2: Diagrama de Cloudbook

1.2. Objetivos

En este trabajo se va a realizar el diseño, desarrollo y prueba de una plataforma que permita a un programador realizar programas distribuidos automáticamente abstrayéndose de los paradigmas tradicionales de programación distribuida. También proporcionará funcionalidad que permita explotar el paralelismo durante la ejecución del programa.

Esto permitirá liberar al programador de realizar un diseño complejo orientado a la distribución, sin renunciar a las ventajas de la computación distribuida o paralela con un coste de aprendizaje de uso muy bajo.

Para lograr este objetivo hay que resolver dos problemas, estos son, el análisis automático del código y la producción a partir de este del código destinado a la distribución, y la distribución misma en la red de máquinas en la que se ejecutará.

1.2.1. Objetivo 1: Análisis y generación automática de código fuente

El primer objetivo de este trabajo responde a la necesidad de generación automática de código orientado a la paralelización y distribución, para aprovechar mejor la potencia de los procesadores, y la distribución automática del mismo.

La plataforma permitirá que, dado un programa, pueda dividirse automáticamente en piezas desplegables (unidades desplegables) que a su vez se distribuyen para su ejecución en un conjunto dado de procesadores, y si el programador quiere, usar etiquetas que permitan aprovechar la paralelización.

Para ello reestructurará el código en función de un criterio definido y lo desplegará en distintos ordenadores que interactúan siguiendo el orden previsto por el programador. Una de las ventajas de este sistema es que el código del programa original no necesita incluir sentencias especiales para su distribución ni es necesario utilizar un lenguaje específico de programación paralela para su desarrollo. Además, las unidades desplegables podrán comunicarse entre ellas sin necesidad de la figura de un orquestador central.

El análisis del código generará un grafo de dependencias entre las funciones del programa original y las distribuirá en distintas unidades desplegables en función del entorno distribuido donde se ejecutará y de distintos criterios como pueden ser el tamaño de las unidades desplegables o la capacidad de comunicación en la red.

Se generará código de forma automática para permitir la comunicación entre distintas unidades desplegables, gestionar variables globales, y permitir la ejecución paralela de ciertas funciones designadas mediante directivas.

1.2.2. Objetivo 2: Distribución de código fuente

El segundo objetivo de este trabajo consiste en el despliegue del código en los distintos ordenadores definiendo conjuntos de estos denominados círculos, que tendrán acceso al código fuente y datos de entrada en un sistema de ficheros distribuido y teniendo en cuenta las necesidades de la red, como el acceso a través de internet a redes privadas con traducción de direcciones (NAT).

Los círculos constarán de distintos ordenadores, cada uno de los cuales ejecutará un agente, que se encargarán conjuntamente de la ejecución del programa distribuido.

Los agentes son procesos que deberán arrancarse en un puerto del ordenador y esperar a conocer qué unidades desplegables les corresponde ejecutar. Además, deberán poder invocar funciones en otros agentes.

Los agentes deberán conocer la ubicación del sistema de ficheros distribuido para poder acceder al código fuente que deben ejecutar, conocer las direcciones del resto de agentes pertenecientes al círculo para poder llamar a sus funciones, y principalmente saber qué unidades desplegables se les han asignado.

El sistema de ficheros distribuido no se tendrá en cuenta en el desarrollo, dejándolo a elección del usuario. La única acción que debe tomar es indicar dónde está para que la plataforma trabaje con el código fuente y datos que allí se encuentran

1.2.3. Objetivo 3: Analizar la plataforma y su uso

Se desarrollará un prototipo funcional con el que validar pruebas de concepto. Esto permitirá analizar el comportamiento de la plataforma, validar las hipótesis planteadas, y medir de forma efectiva las ventajas de la misma.

Otro objetivo dentro de este es conseguir que la plataforma no suponga un desafío para los usuarios y se pueda usar sin un conocimiento específico sobre distribución de software.

Para ello las pruebas se realizarán con código desarrollado para una ejecución no distribuida, ya que la distribución la realizará la plataforma de forma automática.

Las pruebas se probarán con la plataforma, sin ella, con el objetivo de ofrecer un análisis sobre el rendimiento y la facilidad de uso que marquen las fortalezas y debilidades de la plataforma y los casos en los que su uso puede ser más beneficioso.

1.3. Metodología de trabajo

La metodología empleada en este proyecto parte de la definición de un objetivo para el mismo y de un estudio del estado del arte para identificar trabajos relacionados, qué problemas se dan en el ámbito estudiado, y qué se puede aportar al mismo, y así proporcionar una definición más concreta del objetivo del proyecto.

Tras esto, se realizan una serie de hipótesis que permiten el desarrollo iterativo de la plataforma que constituye el proyecto.

El diseño iterativo de la plataforma se realiza en distintas etapas:

1. Propuesta de arquitectura genérica, que es la que se observa en el apartado 2 del capítulo 3 de esta memoria.
2. Se valida la arquitectura desarrollando un prototipo sencillo de cada componente.
3. Sobre cada componente se prueban las distintas hipótesis para poder mostrar su validez y si es necesario aplicar nuevas funcionalidades.

4. Los prototipos de los componentes pueden interactuar para validar el uso en toda la plataforma, y validar así los anteriores pasos y producir propuestas de mejora o nuevas hipótesis para las siguientes iteraciones.

Este diseño iterativo se muestra a lo largo de la descripción de los componentes en el apartado 3 del capítulo 3. Y concluye con el prototipo presentado en este proyecto. Las mejoras futuras de la arquitectura se describen en el capítulo 5.

1.4. Estructura de la memoria

En este trabajo se ha estructurado en los siguientes capítulos:

- Capítulo 1: Introducción y objetivos

En este capítulo se presenta la plataforma sobre la que trata el trabajo introduciendo el problema sobre el que trabaja y la motivación del mismo. También se declaran los objetivos del trabajo y se explica la metodología seguida para llevarlo a cabo.

- Capítulo 2: Estado del arte

En este capítulo se realiza un estudio sobre las tecnologías actuales relacionadas con este trabajo y se discute cuál puede ser nuestra aportación al estado del arte.

- Capítulo 3: Plataforma Cloudbook

Este capítulo se compone de tres partes: La primera, una descripción de las tecnologías usadas para el desarrollo de la plataforma Cloudbook. La segunda, una descripción del proyecto de alto nivel en la que se explica cómo funciona la plataforma. Y la tercera, una descripción pormenorizada de los elementos que componen la plataforma y el funcionamiento de los mismos.

- Capítulo 4: Experimentación y resultados

En este capítulo se presentan distintas pruebas sobre la plataforma desarrollada, que abordan distintos casos de paralelización y distribución. Se discuten asimismo los resultados obtenidos en función de su adecuación a los objetivos propuestos.

- Capítulo 5: Conclusiones y trabajo futuro

En este capítulo se muestran las conclusiones obtenidas en el desarrollo y prueba de la plataforma y se detallan las líneas de trabajo futuro sobre las que seguir implementando la plataforma.

- Anexos: Manual de programación

También se incluye en este trabajo una guía de instalación y uso de la plataforma, junto con recomendaciones de programación.

2. Estado del arte

El estudio de las tecnologías necesarias asociadas a este trabajo se ha dividido en tres secciones según su relación con el proyecto. La primera sección es la referente al estado del arte en el campo de la computación paralela y distribuida. La segunda hace referencia a la compilación, la *trans-compilación* y la generación de código automático. La tercera sección hace referencia a las tecnologías existentes en el mismo ámbito que este proyecto. Por último, en el apartado de conclusiones se discuten las aportaciones y diferencias respecto del estado del arte que este proyecto aborda.

2.1. Computación

En este apartado se describen los tipos de computación que se abordan en el desarrollo de este trabajo.

2.1.1. Computación distribuida

La computación distribuida es el campo de la informática que estudia los sistemas distribuidos, un sistema distribuido es un sistema cuyos componentes colaboran para alcanzar un objetivo común y están en distintas redes, se comunican y coordinan sus acciones mediante paso de mensajes (6).

También se entiende como computación distribuida al tipo de computación que hace uso de sistemas distribuidos para resolver problemas computacionales que se dividen en muchos problemas pequeños que se resuelven en uno o más computadores (7) de forma concurrente o paralela.

La resolución de problemas usando múltiples computadoras comparte muchas similitudes con los problemas concurrentes en una sola computadora, se tienen en cuenta tres puntos de vista principales (8):

- **Algoritmos paralelos en un modelo de memoria compartida**: Todos los procesadores tienen acceso a la memoria compartida, y el programador elige el programa que ejecuta cada procesador. Este modelo es el más cercano a un computador multiprocesador encapsulando la comunicación entre nodos en las instrucciones del programa.

- **Algoritmos paralelos en un modelo de paso de mensajes**: El programador elige la estructura de la red de procesadores y el programa que ejecuta cada uno.
- **Algoritmos distribuidos en un modelo de paso de mensajes**: El programador solo elige el programa, y todos los computadores ejecutan el mismo programa, el sistema debe trabajar correctamente sin tener en cuenta la estructura de la red.

2.1.1.1. Características de la computación distribuida

Las principales características de los sistemas distribuidos son (6):

- Los componentes actúan de forma concurrente.
- Los componentes son entidades autónomas, tienen su propia memoria.
- Carecen de un reloj global, ya que son sistemas autónomos, y necesitan sistemas de sincronización.
- Son independientes a los fallos de los distintos nodos del sistema.
- Los componentes del sistema tienen un conocimiento incompleto del mismo, puede conocer solo una parte de la entrada.
- Pueden comunicarse mediante paso de mensajes, o tener un modelo de memoria compartida.

Los sistemas distribuidos suelen tener varias arquitecturas, como son (9):

- Cliente-servidor: En las que las computadoras clientes se comunican con el servidor para solicitar datos y devolver resultados. Como puede ser la plataforma BOINC, que se describe más adelante.
- Tres niveles: Estas arquitecturas mueven la lógica a un nivel intermedio, de forma que los clientes son más sencillos de implementar, ya que implementan solo la presentación de los datos. Estas arquitecturas las implementan la mayoría de las aplicaciones web.
- N niveles: Estas arquitecturas son las referidas a aplicaciones web que mueven sus peticiones a otros servicios propios o de terceros extendiendo de esta manera las arquitecturas de tres niveles. En este caso se usan para implementar servidores de aplicaciones, como Google App Engine.

- Peer to peer (P2P): En estas arquitecturas las responsabilidades se distribuyen uniformemente y cada computadora puede ser servidor o cliente. Un ejemplo de esta arquitectura es WebTorrent.

El cálculo de la complejidad de los algoritmos distribuidos se tiene que basar también en las operaciones de comunicación, no solamente en la complejidad del programa, ya que el diámetro de la red distribuida cobra una gran importancia, porque hay casos en los que la comunicación puede llevar más tiempo de ejecución que la resolución del algoritmo (10).

El hecho de que los sistemas distribuidos sean por naturaleza asíncronos da lugar a problemas de sincronía que se pueden resolver mediante las siguientes estrategias (10):

- Sincronizadores: Uso de sistemas de sincronización para ejecutar algoritmos síncronos en sistemas asíncronos.
- Relojes lógicos: Proveen un ordenamiento causal de los eventos.
- Sincronización de relojes: Algoritmos que proveen de un reloj global consistente (11).

2.1.1.2. Confusión entre tipos de computación distribuida y paralela

Teniendo en cuenta el objetivo de estos tipos de computación, que es resolver un problema común, hay mucha confusión entre los términos “computación distribuida”, “computación paralela” y “computación concurrente” (11). Un mismo sistema puede considerarse como "paralelo" y "distribuido" ya que los computadores que forman parte de un sistema distribuido se pueden ejecutar simultáneamente en paralelo (10). Es por esto por lo que la computación paralela puede considerarse una forma de computación distribuida estrechamente acoplada (12) y la computación distribuida como una forma de computación paralela acoplada de forma flexible (11).

El criterio más aceptado para distinguir entre términos es [Figura 3]:

- En la computación en paralelo, todos los procesadores pueden tener acceso a una memoria compartida para intercambiar información entre procesadores (13).

- En computación distribuida, cada procesador tiene su propia memoria privada (memoria distribuida). La información se intercambia pasando mensajes entre los procesadores (3).

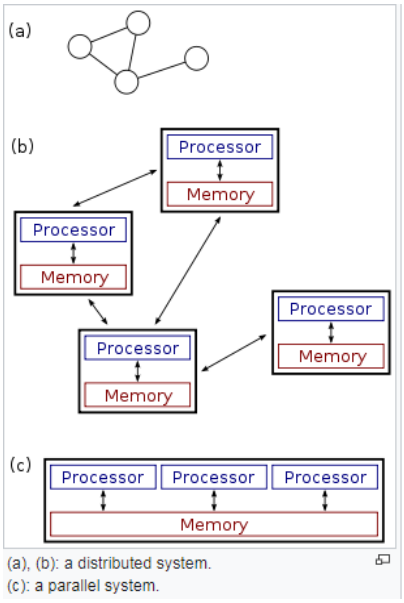


Figura 3: Diferencia entre sistema distribuido y paralelo (14)

2.1.2. Computación paralela

La computación paralela es un tipo de computación en el que muchas operaciones o ejecución de procesos se realizan simultáneamente (15).

Tradicionalmente los algoritmos para resolver un problema se construyen e implementan como un flujo en serie de instrucciones, que se ejecutan en la CPU de una computadora. La computación paralela usa simultáneamente distintos elementos de proceso para resolver un problema, esto es posible separando el problema en distintas partes independientes. Los elementos de proceso pueden ser muy diversos, desde múltiples procesadores en una única computadora, hasta varias computadoras en una red, hardware especializado o una combinación de estas (16).

La computación paralela ha sido usada normalmente en la computación de altas prestaciones (17), pero con la aparición de restricciones físicas que impiden el escalado de frecuencia, y la importancia de la reducción del consumo energético, la programación paralela se ha convertido en el paradigma de computación más usado también en los computadores domésticos (2).

Todos los computadores modernos dan soporte al paralelismo de distintas maneras:

- Procesadores Multi-core (18): Computadores cuyos procesadores incluyen varias unidades de proceso que pueden procesar instrucciones de varias fuentes por ciclo de reloj.
- Multiprocesamiento simétrico (19): Varios procesadores comparten memoria y están conectados por un bus.
- Computadores en clúster (20): Conformados por un grupo de computadoras acopladas que trabajan de forma conjunta.
- Computadores paralelos masivos (21): Un procesador paralelo masivo contiene muchos procesadores conectados, con una escala superior a los computadores en clúster.
- FPGA (22): Una matriz de puertas programables o FPGA es un computador que contiene bloques de lógica cuya conexión puede ser configurada para una tarea dada.
- GPU (23): Unidades de proceso gráfico con capacidad de procesar en paralelo cientos de tareas.
- Computación distribuida (3) y en grid (4): La computación distribuida y en grid ejecuta tareas de forma concurrente que también puede producirse en paralelo.

En función de las comunicaciones entre los distintos procesos paralelos, se distingue el paralelismo de grano fino, en el que la comunicación es muy alta, el paralelismo de grano grueso, en el que hay poca comunicación, y las funciones vergonzosamente paralelas, que nunca o casi nunca se tienen que comunicar (24).

2.1.2.1. Características de la computación paralela

Rendimiento

El rendimiento óptimo de la paralelización es lineal, es decir, duplicar el número de elementos de procesamiento debe reducir a la mitad el tiempo de ejecución, pero en realidad esta aceleración se consigue en muy pocos algoritmos paralelos, la mayoría tienen un aumento del rendimiento casi lineal para pocos elementos de proceso que se estabiliza en un valor constante cuando el número de elementos de procesamiento crece mucho [Figura 4].

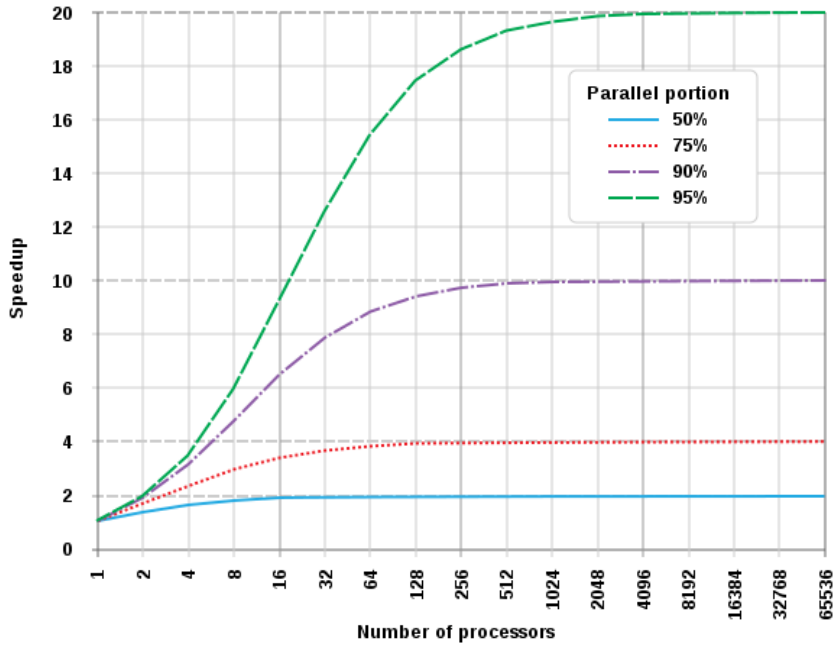


Figura 4: Rendimiento según paralelismo (25)

Este fenómeno se describe mediante la ley de Amdahl, que demuestra que la parte del programa que no puede ser paralelizada limitará la velocidad total de la paralelización (26) según la fórmula:

$$S_{latency}(s) = \frac{1}{1 - p + \frac{p}{s}}$$

- S es la ganancia potencial
- s es la ganancia de la ejecución de la parte paralelizable de la tarea
- p es el porcentaje de la ejecución de la parte paralelizable antes de ser paralelizada.

La ley de Amdahl se aplica para un tamaño de problema fijo. En tareas en las que se dispone de más recursos informáticos, estos tienden a utilizarse para resolver problemas más grandes (conjuntos de datos mayores) y el tiempo que consume la parte paralelizable crece mucho más rápido que el trabajo en serie. En este caso la ley de Gustafson ofrece una evaluación más realista del rendimiento paralelo (27):

$$S_{latency}(s) = 1 - p + sp$$

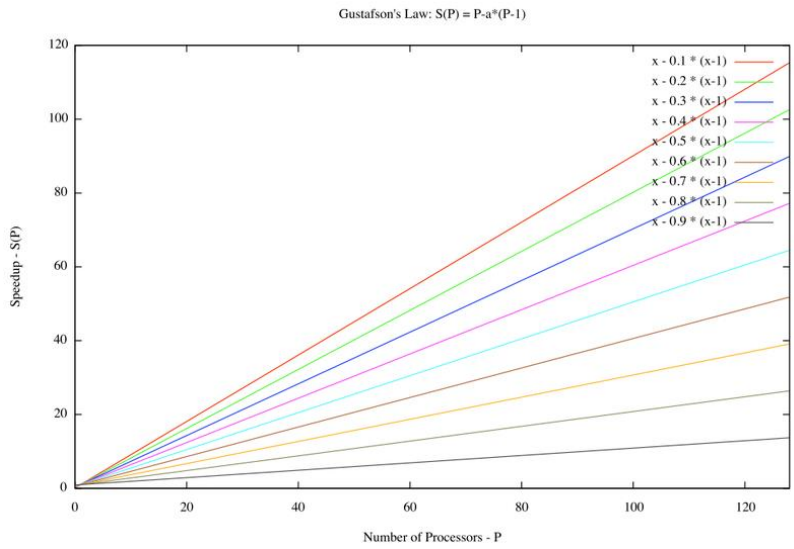


Figura 5: Rendimiento en paralelo según ley de Gustaffson (28)

Tipos de paralelismo

Existen distintos tipos de paralelismo en función las instrucciones en paralelo y acceso a los datos (29):

- Paralelismo a nivel de bit: Se consigue la aceleración a nivel de arquitectura incrementando el tamaño de palabra, y acelerando las operaciones que se pueden realizar en un mismo ciclo ya que el procesador puede procesar más información.
- Paralelismo a nivel de instrucción: Este paralelismo se consigue mediante la división de la ejecución de una instrucción en distintas etapas permitiendo que se puedan separar y poder tener varias instrucciones en distintas etapas en un mismo ciclo.
- Paralelismo a nivel de tarea: Este paralelismo se produce cuando se descompone una tarea en subtareas y se asigna cada una a un procesador para su ejecución, permitiendo realizar cálculos muy diferentes sobre los mismos o distintos conjuntos de datos.

Clasificación de programas. La taxonomía de Flynn

La taxonomía de Flynn es una clasificación de programas y computadoras según su funcionamiento y ejecución de instrucciones y de datos.

Aunque muchas máquinas son híbridos entre estas categorías, este modelo es el más usado debido a su sencillez, porque es muy fácil de entender y da una buena primera aproximación al funcionamiento de los programas paralelos (5).

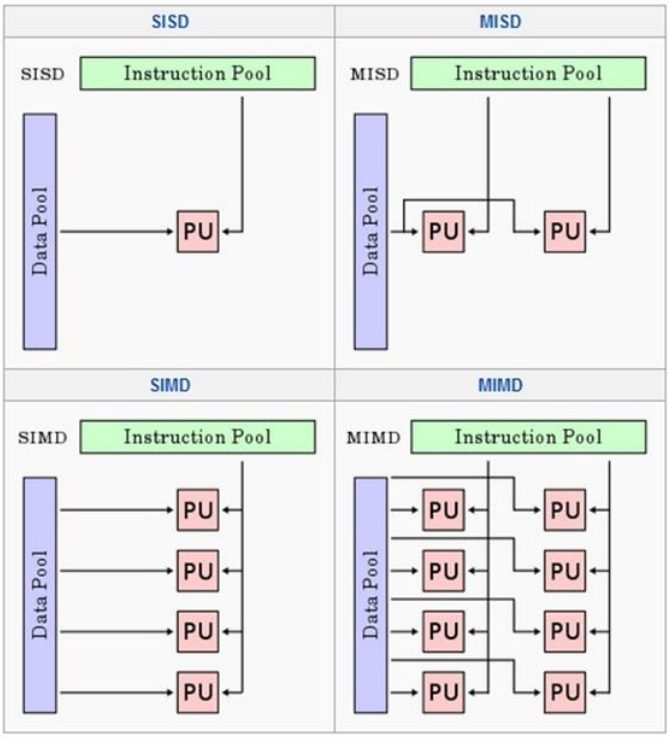


Figura 6: Taxonomía de Flynn (30)

- SISD (Single Instruction Single Data): Es cómo funcionan los programas secuenciales, se realiza una instrucción cada vez sobre un conjunto de datos.
- MISD (Multiple Instruction Single Data): Aquí se clasifican los programas que realizan varias instrucciones en paralelo sobre un mismo conjunto de datos.
- SIMD (Single Instruction Multiple Data): Los programas que realizan la misma operación sobre distintos conjuntos de datos, donde más se usa es en el procesado de señales.
- MIMD (Multiple Instruction Multiple Data): Los programas paralelos más comunes, realizan a la vez distintas instrucciones sobre distintos conjuntos de datos.

2.1.2.2. Paralelismo en lenguajes de programación

Los lenguajes de programación de alto nivel incorporan diversas técnicas para obtener paralelismo. En función del paradigma de programación empleado podemos dividir estas técnicas:

Programación imperativa: Este es el paradigma as natural de programación ya que establece una abstracción de alto nivel de la máquina en la que se ejecuta, los lenguajes imperativos programan la lógica y el control del algoritmo, se centran en "cómo se hace" un programa. El paralelismo se establece mediante pragmas o cambios en el lenguaje, como se refleja en las tecnologías, como OpenMP o CUDA, que se verán más adelante.

Programación declarativa: El paradigma de la programación declarativa en el que el programa se centra en la lógica del algoritmo, en "que hace" el programa. Dentro de los lenguajes declarativos está el paradigma de la programación funcional, en el que las funciones se declaran como una abstracción de funciones matemáticas. Permite explotar el paralelismo implícito sobre un conjunto de datos. Un ejemplo de esto es la herramienta Spark, cuyo lenguaje oficial es Scala, que es un lenguaje funcional y permite realizar desarrollos sobre *big data* de forma paralela.

Como se ve en la figura 7, una función escrita en Scala para Spark consiste en una serie de operaciones sobre unos datos, cada transformación de los datos es fácilmente paralelizable ya que solo necesita los datos a la entrada de la función y guardar los nuevos datos una vez termine, es paralelismo de grano grueso con muy poca comunicación.

```
sparkContext.textFile("hdfs://...")
            .flatMap(line => line.split(" "))
            .map(word => (word, 1)).reduceByKey(_ + _)
            .saveAsTextFile("hdfs://...")
```

Figura 7: Ejemplo de programación funcional

2.1.3. Computación grid

La computación grid es un paradigma computacional que se encuentra dentro de la computación distribuida, y se basa en el uso de recursos computacionales ampliamente distribuidos para alcanzar un objetivo común. La computación grid se puede ver también como un tipo de computación paralela basada en computadores completos dentro de una red (31).

Un sistema grid se define como una infraestructura hardware y software que provee de acceso fiable, consistente generalizado y económico a capacidades computacionales de alta gama. Los sistemas grid pueden conformar supercomputadores distribuidos compuestos de muchas computadoras distribuidas en una red y débilmente acopladas que funcionan conjuntamente para realizar tareas pesadas (4).

2.1.3.1. Características de la computación grid

Las principales características de los sistemas grid son (32):

- Coordina recursos de forma no centralizada: No hay un sistema de gestión local, puesto que un grid se compone de recursos heterogéneos.
- Usa protocolos e interfaces abiertos y de propósito general: Un grid no es específico para una aplicación, es por eso que usa protocolos e interfaces multipropósito.
- Un grid ofrece un resultado de calidad: Un grid permite usar los recursos que lo constituyen de manera no coordinada para cumplir con necesidades complejas, por lo que la utilidad del sistema combinado es mayor que la suma de sus partes.

Los sistemas grid se pueden clasificar en 3 tipos (4):

- Grid computacional: Los sistemas aprovechan las máquinas que los conforman para obtener mayor capacidad de cómputo que cualquier máquina del sistema.
- Grid de datos: Los sistemas que proveen una infraestructura hardware y software para sintetizar repositorios de datos distribuidos en una red.
- Grid como servicio: Referido a los sistemas grid que proveen servicios que no provee ninguna máquina por sí sola.

2.1.3.2. Diferencias entre grid y otros tipos de computación distribuida

La computación grid tiene muchos puntos en común con otros paradigmas dentro de la computación distribuida. De hecho, es la parte fundamental sobre la que se asientan otras tecnologías como la computación en la nube, cuya principal diferencia se basa en que esta está orientada a proporcionar servicios y recursos de una forma abstracta, en lugar de recursos computacionales (33).

También tiene relación con otros dominios más orientados a aplicaciones que a servicios, como la supercomputación, donde los sistemas en grid ofrecen menor rendimiento (33). Un sistema grid se distingue de un sistema convencional de computación de altas prestaciones como puede ser la computación en clúster en que en los sistemas grid cada nodo realiza una tarea o ejecuta una aplicación distinta o independiente, y que los sistemas grid tienden a ser más heterogéneos y desacoplados que los sistemas en clúster (34).

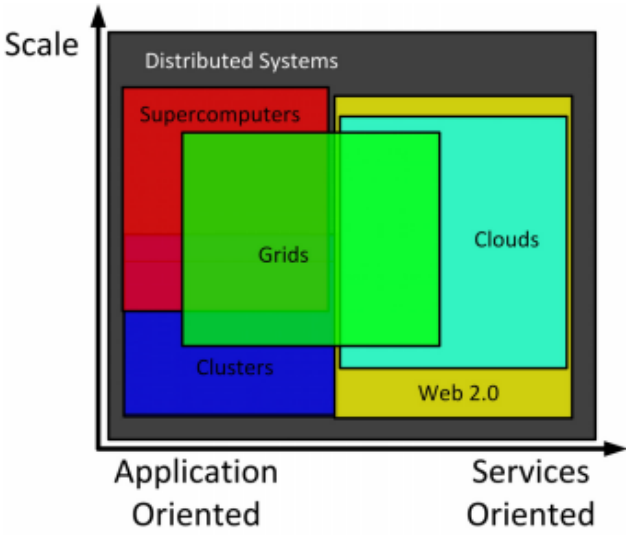


Figura 8: Clasificación de sistemas distribuidos (32)

2.1.4. Sistemas de ficheros distribuidos

Un sistema de ficheros distribuido es un sistema de ficheros que permite a los programas almacenar y acceder a archivos de forma transparente, como si fuesen archivos locales, con la diferencia de que se accede desde cualquier computadora de la red o sistema distribuido. El rendimiento y la fiabilidad de los accesos a los archivos almacenados en un servidor debe ser comparable a la de los archivos almacenados en sistemas de ficheros locales (35).

2.1.4.1. Características de los sistemas de ficheros distribuidos

Las principales características que deben reunir los sistemas de ficheros distribuidos son (35):

- Transparencia: Esta es la principal característica de los sistemas de ficheros distribuidos, ya que el usuario no debe saber dónde están ubicados los ficheros, ni debe afectar al rendimiento de los programas, y tampoco debe ser consciente de los cambios en el sistema.

- Concurrencia y consistencia: Los cambios que hace un cliente no deben interferir con los cambios que puede estar realizando otro cliente a los mismos ficheros en ese momento y las distintas copias replicadas o cacheadas deben ser consistentes teniendo en cuenta la propagación de las modificaciones.
- Replicación: Los sistemas de ficheros que permiten la replicación tienen dos ventajas, la distribución de la carga a la hora de proveer a los clientes y la tolerancia a fallos, algunos sistemas tienen un sistema de replicación parcial mediante el cacheo de ficheros local.
- Sistemas heterogéneos: La interfaz del sistema de ficheros distribuido ha de permitir que clientes y servidor están en distintos sistemas.
- Tolerancia a fallos: Es esencial que el sistema de ficheros distribuido continúe su servicio a pesar de las posibles caídas de servidores y clientes.
- Eficiencia: El nivel de rendimiento de los sistemas de ficheros distribuidos son comparables al de los sistemas de ficheros convencionales.
- Seguridad: En los sistemas de ficheros distribuidos existe la necesidad de autenticar a los clientes y proteger los contenidos de los mensajes mediante firmas digitales y encriptación de los datos.

Las distintas arquitecturas en las que se puede basar un sistema de ficheros distribuidos son (35):

- Cliente-servidor: Este tipo de sistemas de ficheros, como NFS de Sun Microsystems, incorpora un protocolo de comunicación que permite a los clientes acceder a los ficheros de un servidor permitiendo el acceso por diversos procesos.
- Basado en clúster: Este tipo de sistema de ficheros, como Google File System tiene una arquitectura basada en un maestro único junto con servidores de fragmentos de un tamaño determinado, esto permite que un único maestro controle cientos de servidores de forma sencilla.
- Arquitectura simétrica: Esta arquitectura es la basada en el modelo P2P se basa en una tabla hash distribuida para repartir los datos, combinado con un mecanismo de búsqueda basado en claves. En este tipo de arquitectura los clientes también contienen el código de administración de los metadatos de los ficheros.

- Arquitectura asimétrica: Este tipo de sistemas de ficheros contiene uno o más gestores de metadatos que mantienen el sistema de ficheros y sus estructuras asociadas. Ejemplos de esto pueden ser Lustre o Panasas ActiveScale.
- Arquitectura paralela: Estos sistemas de ficheros distribuidos separan los bloques de datos en paralelo en múltiples dispositivos o servidores de almacenamiento y permiten a las aplicaciones paralelas acceder a los nodos realizando lecturas y escrituras concurrentes.

2.2. Procesadores de lenguajes

En este apartado se describen las tecnologías y procesos que se usan para compilar lenguajes, o procesarlos de forma automática.

2.2.1. Procesadores de lenguajes y compiladores

Un compilador es un programa que traduce el código fuente de un programa escrito en un lenguaje de programación a otro lenguaje, cuyo uso principal es para traducir código de un lenguaje de alto nivel a uno de más bajo nivel (ensamblador, código objeto, o código máquina) para crear un programa ejecutable. Que se ejecuta con una entrada que introduce el usuario (36).

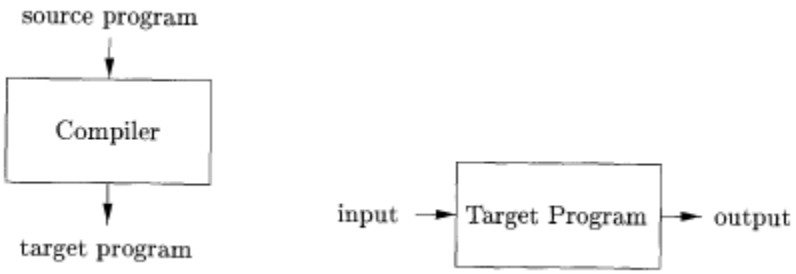


Figure 9: Funcionamiento de un compilador (36)

Un intérprete es otro tipo de procesador de lenguajes, pero en lugar de producir un programa ejecutable tras la traducción el intérprete directamente ejecuta las operaciones especificadas en el código fuente sobre las entradas del usuario.

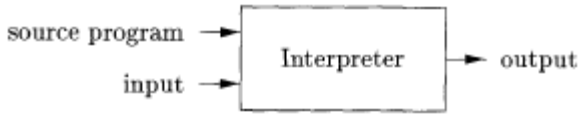


Figure 10: Funcionamiento de un intérprete (36)

2.2.1.1. Compiladores fuente a fuente y otros tipos de compiladores

Un compilador fuente a fuente o un *trans-compilador* es un tipo específico de compilador que traduce entre lenguajes de alto nivel. Este tipo de compiladores se suelen usar en herramientas de paralelismo automático que toman como entrada un lenguaje y producen un código en ese mismo lenguaje con nuevas capacidades de código paralelo (como OpenMP) u otras construcciones del lenguaje (declaraciones generales en Fortran), como se verá más adelante que hacen tecnologías como Pluto.

También se usa para traducir código heredado a la siguiente versión de un lenguaje de programación o para conversión entre distintas especificaciones cuando el refactorizado del código es muy complejo.

Además de los compiladores fuente a fuente, existen otros tipos de compiladores (36):

- Compilador cruzado: Produce un programa compilado que puede ejecutarse en una computadora cuya CPU o sistema operativo es distinto del original en el que se realiza la compilación.
- *Bootstrap compiler*: Es un compilador escrito en el lenguaje que va a compilar.
- Descompilador: Un programa que convierte un lenguaje de bajo nivel a un nivel más alto.
- También entran dentro de este campo los *reescritores* de lenguaje que traducen expresiones en un lenguaje sin cambiar el mismo. Y el término generador de compiladores o *compiler-compiler* que se refiere a las herramientas usadas para crear analizadores sintácticos o *parsers*.

2.2.1.2. Funcionamiento de un compilador

Los compiladores realizan las siguientes operaciones (36):

- Análisis léxico: Es la primera fase de un compilador, el analizador léxico lee el código fuente y agrupa los caracteres en lexemas, y por cada lexema produce un token

<token_name, token_value>

- El nombre del token es un símbolo definido durante el análisis sintáctico que refleja la naturaleza del lexema (identificador, palabra clave, número entero...) el segundo campo es el valor y se apunta en la tabla de símbolos, para usarlo más adelante en el análisis semántico y generación de código.
- Análisis sintáctico: Sobre los tokens del paso anterior, crea una estructura arbórea que representa el programa. Esta estructura es el árbol sintáctico.
- Análisis semántico: Sobre el árbol sintáctico y la información de la tabla de símbolos comprueba el código fuente para comprobar la consistencia semántica con la definición del lenguaje. En este paso se realiza el chequeado de tipos, para comprobar que es coherente el uso de los operadores.
- Generación de código intermedio: En algunos compiladores se produce código explícito de bajo nivel que sea fácil de producir y convertir en código máquina para representar el resultado obtenido del árbol semántico.
- Optimización de código: En este paso se optimiza el código intermedio generado en el paso anterior. Hay muchas formas de optimizar el código, ya sea para que sea más rápido, ocupe menos, o consuma menos.
- Generación de código: En este último paso se traduce el programa al lenguaje objetivo, en caso de ser código máquina, la elección de registros para guardar las variables y de direcciones es especialmente importante.

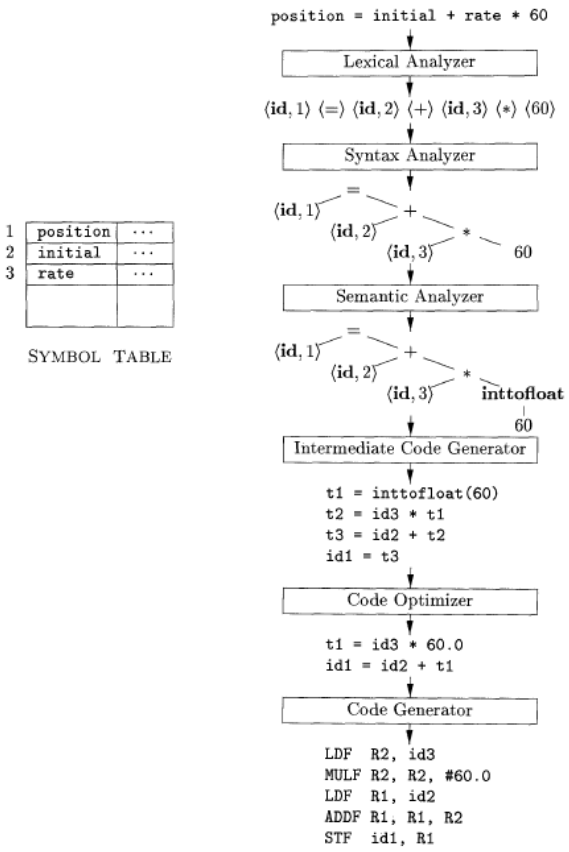


Figure 11: Ejemplo funcionamiento de un compilador (36)

2.2.2. Paralelización automática

La paralelización automática es el proceso por el que se convierte código secuencial en código multihilo o vectorial que pueda usar múltiples procesadores simultáneamente. El objetivo de esta forma de paralelización es liberar a los programadores del proceso de paralelización manual que es muy complejo y propenso a errores (5). Aunque se han conseguido avances en este campo sigue siendo uno de los grandes retos de los compiladores, debido al complejo análisis que hay que realizar y factores desconocidos como el tamaño de los datos de entrada durante la compilación (37).

Las técnicas de paralelización se basan en herramientas de tiempo de compilación o tiempo de ejecución en las que el usuario debe identificar el código de paralelización con construcciones especiales (directivas o etiquetas). El compilador identifica estas construcciones y analiza el código que se va a paralelizar, por lo que es necesario disponer de una herramienta automática que convierta el código secuencial en paralelo (38).

Las principales dificultades de la paralelización automática en los compiladores son (39):

- Análisis de dependencias: Detectar dependencias en el código cuando usa punteros, recursión, o llamadas indirectas es muy difícil de detectar en tiempo de compilación.
- El número de iteraciones de un bucle es desconocido.
- Los accesos a recursos globales son difíciles de coordinar en términos de asignación de memoria.
- Los algoritmos irregulares que usan indirección dependiente de la entrada interfieren con el análisis y la optimización en tiempo de compilación.

Para solventar estas dificultades que se dan en la paralelización automática completa, existen aproximaciones que permiten desarrollar código paralelo de calidad sin ser completamente automático:

- Permitir a los programadores añadir "pistas" en forma de directivas o etiquetas que guían la paralelización. Como hace la tecnología OpenMP incluyendo pragmas.
- Construir sistemas interactivos entre el programador y las herramientas de paralelización o compiladores, siendo los principales ejemplos de esta aproximación Vector Fabrics, Pareon, y SUIF Explorer
- Hardware que permite hacer *multithreading* especulativo.

La estructura en la que más se hace hincapié en la auto paralelización es en los bucles, ya que en ellos pasa la mayor parte de la ejecución de un programa. Hay dos aproximaciones en la paralelización de bucles: *pipelined multi-threading* y *cyclic multi-threading* (40). Se diferencian en la aproximación sobre los hilos que generan para tratar las iteraciones, ya sea por orden de ejecución o por operaciones.

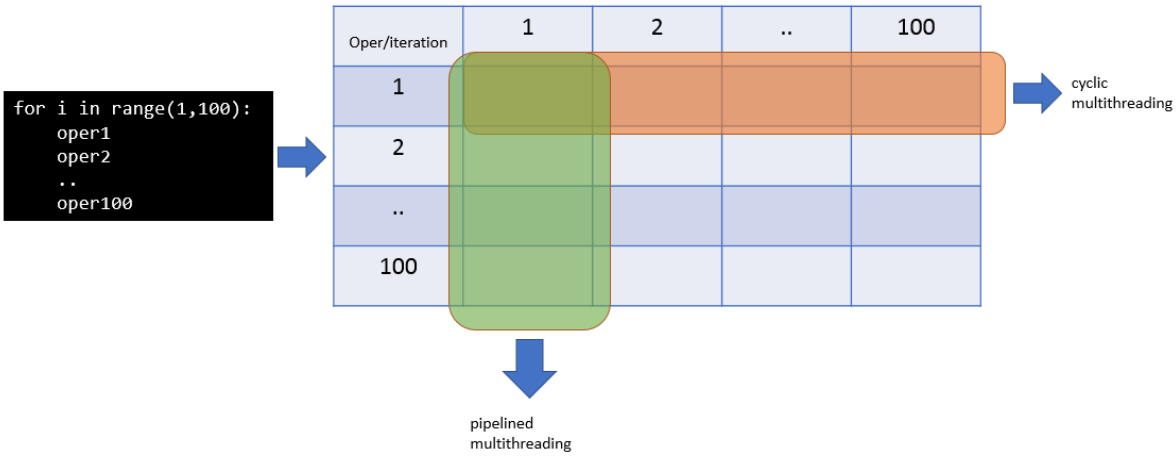


Figure 12: Paralelismo automático sobre bucles

2.3. Tecnologías existentes

En este apartado se describen las tecnologías actuales que trabajan en el ámbito de la computación distribuida y paralela.

2.3.1. Sistemas paralelos y distribuidos

En este apartado se estudian los principales sistemas de cómputo que usan los paradigmas de computación paralela, distribuida o grid.

2.3.1.1. Charm++

Charm++ es un framework de programación paralelo en el lenguaje C++ desarrollado por el laboratorio de programación paralela de la universidad de Illinois. El objetivo de Charm++ es el de mejorar la productividad de los programadores proporcionándoles una abstracción de alto nivel para permitir paralelismo en los programas y que pueda usarse en una gran variedad de plataformas hardware (41).

Las principales características de charm++ son:

- Objetos migrables: Charm++ está basado en la orientación a objetos, y son estos sobre los que se realiza el desarrollo distribuido y la ejecución paralela.
- Invocaciones asíncronas: La comunicación entre el sistema distribuido es transparente y funciona como la invocación a un método de objetos remotos.

- Entorno de ejecución adaptativo: El entorno de ejecución orquesta la ejecución, el diseñador no se preocupa por los hilos o procesadores que intervienen.

Modelo de programación

Los programas escritos en Charm++ se dividen en objetos llamados *chares* que cooperan entre ellos mediante paso de mensajes. Cuando un programador invoca un método de un objeto, el entorno de ejecución de Charm++ envía un mensaje al objeto invocado que reside, bien en el procesador de la propia máquina o en un procesador remoto que se ejecuta en paralelo y este mensaje inicia la ejecución de código del *chare* que se gestiona de forma asíncrona.

Los *chares* de un programa se asignan a procesadores físicos en el entorno de ejecución dinámico, esta asignación es transparente al usuario lo que permite al entorno de ejecución cambiar de forma dinámica la ubicación de los *chares*, balancear la carga, tener tolerancia a fallos, y permitir disminuir o aumentar el número de procesadores usado por un programa paralelo de forma automática.

Paso de mensajes

Charm++ usa una implementación de MPI (Message Passing Interface) adaptativa, conocida como AMPI, que funciona encapsulando los procesos MPI dentro de los *chares* permitiéndoles usar las ventajas del entorno de ejecución de Charm++, sin cambiar o cambiando muy poco la parte de MPI del programa.

CharmPy (42)

CharmPy es un modelo de programación paralela basado en Python y construido sobre el entorno de ejecución de Charm++. Está basado en los mismos paradigmas que Charm++, objetos migrables, invocaciones y paso de mensaje asíncronos, y el entorno de ejecución adaptativo. Está diseñado con el objetivo de ser más sencillo de usar que Charm++, incluye su propia sintaxis, y que a la vez disfrute de las mismas ventajas.

2.3.1.2. BOINC

La infraestructura abierta de Berkeley para la computación en red (BOINC) (43) es una plataforma de computación distribuida de altas prestaciones (*high throughput computing*) basada en la computación grid, que se usa para la colaboración voluntaria en la investigación en diversos campos. El objetivo de este proyecto es obtener una capacidad de computación similar a la de un supercomputador usando los computadores personales de los voluntarios.

Las principales características de BOINC son:

- Está basado en una arquitectura cliente-servidor que se comunican para distribuir, procesar y retornar unidades de trabajo.
- Para la planificación de la ejecución usa el protocolo RPC de llamadas a procedimiento remoto. La gestión de ficheros se realiza mediante HTTP.
- Usa un sistema de créditos para evitar trampas en la validación de resultados, se reciben en función de las computaciones válidas que realiza cada cliente y permite medir la contribución de los voluntarios. Esto es posible mediante un banco de pruebas que realizan los voluntarios cuyo objetivo no es medir la velocidad si no la validez.

Modelo de programación

BOINC tiene una arquitectura cliente-servidor muy rígida, proporciona la arquitectura y los usuarios programan la parte de cálculo que tiene que realizar el cliente, y la que tiene que pedir el servidor (véase Figura 13).

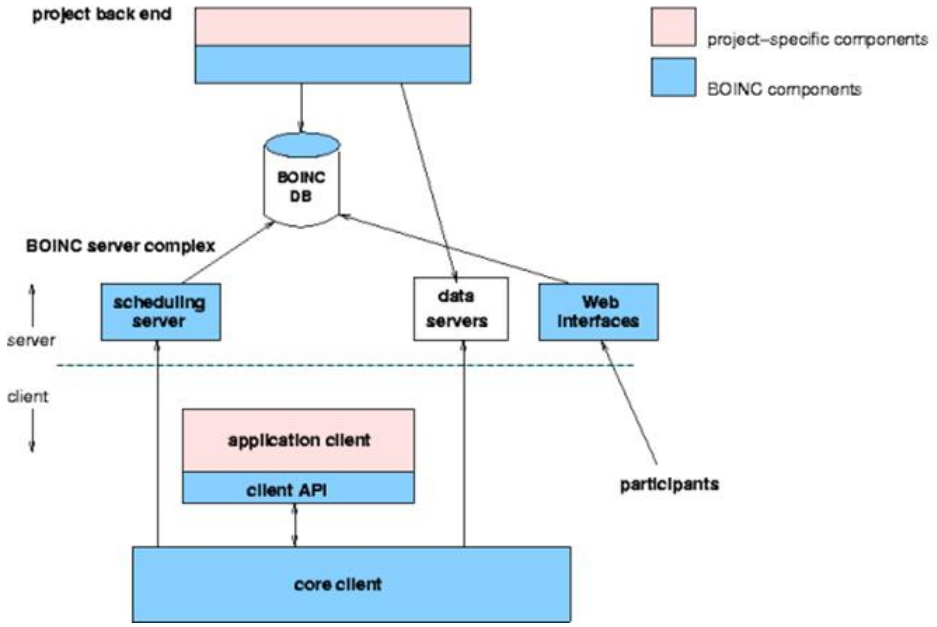


Figure 13: Arquitectura BOINC (44)

2.3.1.3. Spark

Apache Spark es un framework de computación paralela basado en la computación en clúster, su objetivo es proporcionar una interfaz de programación de clústeres completos

aprovechando el paralelismo implícito de los datos, se usa sobre todo para tratar problemas de Big Data. Fue desarrollado en la universidad de California y posteriormente donado a la Apache Software Foundation. (45)

Las principales características de Apache Spark son:

- Escalabilidad: La capacidad de cómputo aumenta cuando se añaden nuevos equipos.
- Resistente a fallos: Los nodos pueden caer, pero no se aborta el cómputo, se recuperan los datos perdidos.
- Flexibilidad: Soporta más operaciones que otros esquemas de cómputo anteriores como MapReduce.
- Eficiencia: Trabaja sobre datos en memoria, para reducir los tiempos de acceso a los mismos.

Modelo de programación

El lenguaje oficial de Spark es Scala, que es un lenguaje funcional, aunque también hay interfaces para Python o Java, es de la programación funcional donde explota el paralelismo de las operaciones.

Usa un esquema maestro-esclavo donde un programa *driver* se conecta a un clúster de *workers*, que almacenan los datos sobre los que se ejecutan las operaciones del programa *driver*.

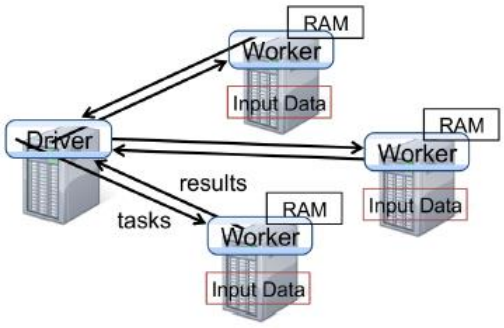


Figure 14: Arquitectura Spark (46)

El concepto fundamental de Spark, y en el que se expresan todos sus cómputos es el Resilient Distributed Dataset (RDD) que es una colección de elementos que se reparte entre los computadores del clúster. las principales características de los RDDs son:

- Resiliencia: Si un nodo deja de funcionar, el fragmento de RDD perdido se recalcula de forma transparente.
- Las particiones de un RDD se calculan automáticamente (aunque se deja la opción de indicar explícitamente el número de particiones)
- Las operaciones se realizan en paralelo sobre todos los fragmentos de RDD.
- Los RDDs son inmutables, no se modifican, se recalculan con nuevas operaciones.
- Las operaciones que soportan los RDDs son: Transformaciones, que realizan algún cálculo y crean un nuevo RDD; Acciones, devuelven un valor al *driver* o vuelcan los datos a la unidad de almacenamiento.

2.3.2. Herramientas de paralelización

En este apartado se presentan distintas herramientas dedicadas a ofrecer paralelismo sobre lenguajes de programación.

2.3.2.1. OpenMP

OpenMP es una API para la programación multiproceso con memoria compartida, su objetivo es añadir concurrencia a programas escritos en C, C++ y Fortran. Se compone de un conjunto de directivas de compilador, rutinas de biblioteca y variables de entorno que se tratan en tiempo de ejecución (47).

Se puede ejecutar en modelos de programación distribuida, usando MPI en un clúster de computadoras, y tiene extensiones para sistemas de memoria distribuida.

Las principales características de OpenMP son:

- Portabilidad: OpenMP está diseñado para la mayoría de las arquitecturas y compiladores del mercado.
- Escalabilidad: OpenMP proporciona una interfaz de programación destinada a computadoras de escritorio tanto como para supercomputadoras.
- Modelo de ejecución basado en *fork-join*: Cuando openMP marca un bloque como paralelo se incluye automáticamente una sincronización de todo el bloque, esto es el modelo *fork-join* en el que se separa la ejecución en hilos, pero se espera a que todos terminen para recolectar los resultados.

- Desde la versión 4.0 permite paralelismo asíncrono: Mediante el paralelismo basado en tareas que se regulan mediante dependencias, reflejadas por un grafo.

Modelo de programación

La programación con OpenMP se basa en el uso de directivas para definir las partes del código del programador que se pueden realizar de forma paralela, en este sentido se definen directivas para declarar secciones paralelas en la que se pueden ejecutar todos los hilos en paralelo, sección paralelas en las que solo puede entrar un único hilo, secciones críticas en las que solo puede entrar un hilo, barreras, o definir tareas, que se pueden ejecutar de forma asíncrona.

Además de definir secciones de código como paralelas, OpenMp ofrece cláusulas para indicar la visibilidad de los datos durante la ejecución paralela, es decir, si los hilos tienen copias locales de los datos o trabajan con variables compartidas.

Por último, también ofrece cláusulas de planificación para la ejecución de los hilos, y de dependencias para las tareas.

2.3.2.2. CUDA

CUDA (Compute Unified Device Architecture) es una plataforma de computación en paralelo para programar con GPUs de nVidia. La plataforma incluye un compilador y herramientas de desarrollo, se puede usar desde Python o Java además de C y C++ que son los lenguajes para los que está diseñado (48).

El objetivo de CUDA es explotar el paralelismo que ofrecen sus GPUs con un gran número de núcleos, que permite que se lance un gran número de hilos simultáneos que pueden realizar tareas independientes.

Las principales características de CUDA son:

- CUDA permite realizar lecturas dispersas, de cualquier posición de memoria.
- Estructura de memoria compartida: Permite usar memoria para compartir entre hilos que por su tamaño y velocidad se puede usar como una memoria caché.
- Tiene un nivel de recursividad muy limitado (profundidad máxima de 24), punteros a función variables estáticas, o funciones con número de parámetros variable.

- SIMT: Single Instruction Multiple Threads: Permite el paralelismo en todos los hilos ejecutando la misma instrucción en múltiples datos sobre los que trabaja cada uno.

Modelo de programación

Para programar en CUDA es necesario comprender su arquitectura formada por:

- Device: La GPU.
- Cudacore: Núcleos de la gpu, cada uno puede lanzar un *thread*.
- Agrupación lógica: Bloque. Varios bloques forman un grid.
- Agrupación física: Multiprocesadores.
- Hay distintos tipos de memoria para distintas funcionalidades, de la más rápida a la más lenta, estas son: *register*, *shared*, *constante*, de *texturas*, y *global*.

La forma más eficiente de programar con CUDA, es dividir el problema en bloques. Las funciones en la GPU son las de tipo *kernel,* se lanzan desde la CPU y se ejecutan en la GPU. Usan una sintaxis de llamada propia:

Mykernel <<numBlocks, NumThreads_per_block>>

En la llamada a la función está implícito el paralelismo, la GPU, estructurada en un grid de bloques con un número determinado de threads lanza la función en todos a la vez con un offset de datos dado. Cada cudacore ejecuta un thread, pudiéndose lanzar miles simultáneamente. Los cores disponibles se reparten entre todos los threads.

Las funciones *kernel* no son bloqueantes, retornan control a la CPU inmediatamente. Debido a que no son bloqueantes, las funciones *kernel* no retornan nada, no hay *callback*. Hay funciones para sincronizar (esperar) la ejecución de todos los *threads* de un bloque y de un grid.

Mediante CUDA es fácil lograr que un programa lance del orden de 100.000 *threads* en una tarjeta gráfica de gama media, donde los *threads* activos simultáneos pueden ser del orden de 2000. Los hilos se agrupan en *warps* (conjuntos de 32 hilos) y cuando la instrucción que deben ejecutar es la misma, el hardware permite que en el mismo ciclo se ejecute la misma instrucción sobre distintos datos en los 32 hilos. Cuando hay divergencia de instrucciones a ejecutar en los *threads* de un *warp,* la velocidad baja, pues el "*instruction dispatcher*" debe entregar una a una la instrucción a ejecutar a cada *thread*. Este funcionamiento implica que se

requiere mucha experiencia para producir programas que aprovechen al máximo los recursos de la GPU.

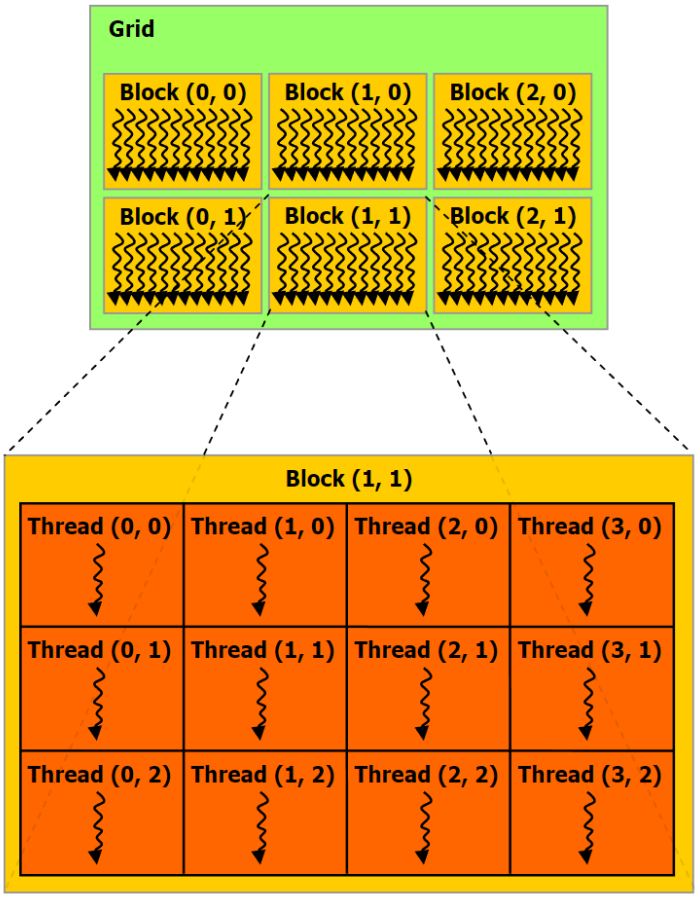


Figura 15: Jerarquía en la programación en CUDA (49)

2.3.2.3. Pluto

Pluto es una herramienta de paralelización automática para el lenguaje C basada en el modelo poliédrico. Pluto realiza un proceso de *trans-compilación* sobre el código fuente, para producir código con paralelismo de grano grueso y localidad de datos, en C con OpenMP (50) (51).

Para conseguir el paralelismo de grano grueso se particiona el espacio de iteración que nos da el modelo poliédrico para poder ejecutar distintos bloques concurrentemente en distintos procesadores con un nivel de intercomunicación muy bajo. El paralelismo con localidad de datos requiere agrupar puntos del espacio de iteración en bloques muy pequeños para que se puedan ubicar en memorias más rápidas y pequeñas (cache).

Funcionamiento. El modelo poliédrico

Dado un programa, cada instancia dinámica de una declaración se define por su vector de iteración que contiene valores para los índices de los bucles en los que se encuentra la declaración. Cuando los límites de un bucle son combinaciones lineales respecto de los bucles más externos, el conjunto de vectores de iteración de una declaración se considera un politopo, es decir, una generalización del poliedro definido por los distintos vectores de iteración.

Cada dependencia entre declaraciones se representa como una arista, que a su vez es un poliedro que representa la dependencia y el orden de ejecución.

Pluto trabaja mediante transformaciones en esto modelo, que pueden modificar el poliedro y el orden de ejecución.

Las dependencias de los datos también se basan en el modelo poliédrico, se obtienen las dependencias a partir de un análisis del flujo de datos y se genera el poliedro de dependencias de datos, formado por vectores de dependencias entre distintas instancias.

2.3.3. Sistemas de ficheros distribuidos

2.3.3.1. Sistema de archivos en red (NFS)

Network File System (NFS) es un protocolo de nivel de aplicación que permite a los usuarios montar sistemas de archivos sobre la red e interactuar con los mismos como si fuesen locales. Sus principales características son (52) (53):

- Arquitectura cliente-servidor: El servidor contiene los datos a los que acceden los clientes de forma transparente.
- Los servidores NFS son sin estado, lo que permite que el servidor no trate los fallos de los clientes.
- La comunicación cliente-servidor se basa en tres protocolos: RPC para la comunicación cliente-servidor, NFS protocolo para las operaciones sobre ficheros, y un protocolo para operaciones de montar/desmontar directorios

Funcionamiento

NFS usa la interfaz de Unix de Virtual File System para el acceso a ficheros, ya que permite el acceso a ficheros remotos, como si fuesen locales. Por cada fichero abierto mantiene un v-

nodo (i-nodo virtual), si representa un fichero local, apunta directamente al i-nodo correspondiente, en caso de que el fichero sea remoto, apunta a un r-nodo (i-nodo remoto) que almacena un manejador de fichero que usará el cliente NFS en la llamada al servidor.

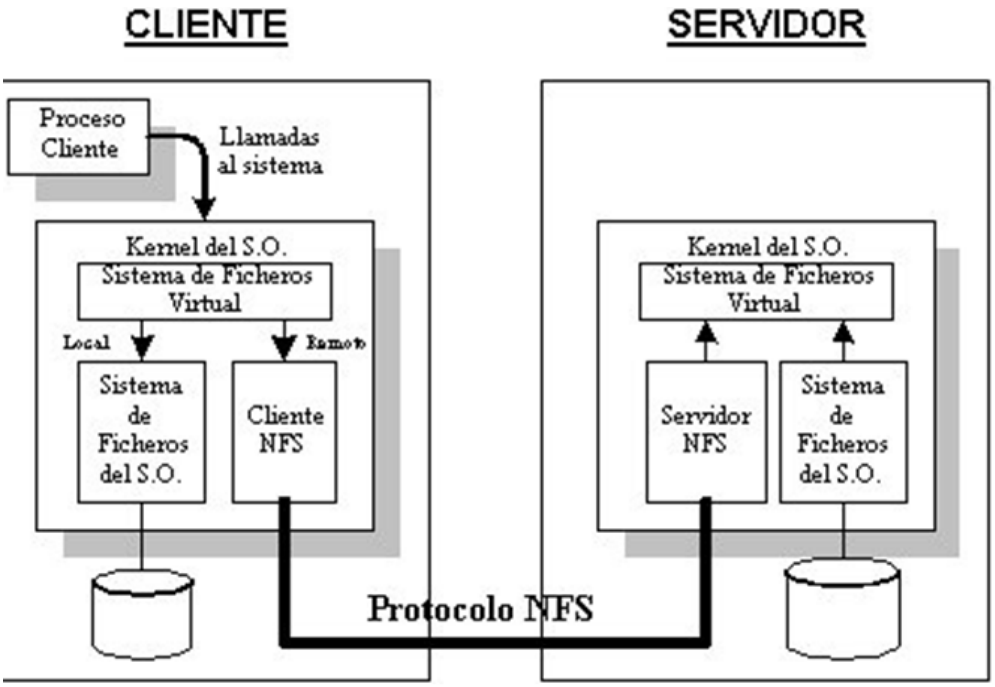


Figura 16: Arquitectura NFS (54)

NFS trabaja con bloques grandes (8Kbytes) por lo que proporciona una lectura anticipada, y la política de escritura es *write-through* por bloque, es decir, actualiza el servidor cuando tiene un bloque completo.

El cliente valida periódicamente si un bloque cargado en su caché ha cambiado, actualizando su copia en este caso. Esto es uno de los problemas de mantenimiento de la consistencia en NFS, ya que si se reduce el tiempo de validación se sobrecarga la red con peticiones RPC *getattr*.

No está diseñado con mecanismos de exclusión mutua, por lo que es necesario un mecanismo de terceros si se quiere bloquear el acceso a ficheros de terceros.

2.3.3.2. Hadoop File System

HDFS es el sistema de ficheros distribuido de Hadoop, diseñado para ejecutarse en computadoras de bajo coste o de uso doméstico. Está especialmente orientado al almacenaje

de archivos de un clúster de máquinas, y a la lectura de datos en stream más que a la interactividad. Las principales características de HDFS son (55):

- Está diseñado para almacenar ficheros grandes que se leerán de forma secuencial.
- Está orientado a clústeres, por lo que escala con facilidad, al añadir nuevos nodos.
- Proporciona redundancia, almacena varias copias de los datos para hacerlo más robusto y tolerante a fallos.
- Almacena los ficheros en bloques muy grandes (128 MB) lo que permite que se puedan realizar lecturas paralelas de un fichero, ya que los bloques no tienen por qué estar en la misma máquina.

Funcionamiento

Las máquinas que implementan HDFS pueden ser de dos tipos (véase Figura 17):

- Namenode: Almacena la información necesaria sobre la estructura de directorios y ficheros, y contiene información sobre los distintos bloques que forman los ficheros y la localización de estos. Obtiene esta información de los datanodes cuando arranca el sistema de ficheros. Se encarga también de balancear la información en los datanodes.
- Datanode: Su función es almacenar los bloques que componen cada fichero, y proporcionarlos a quien lo solicite, ya sea un cliente o un namenode.

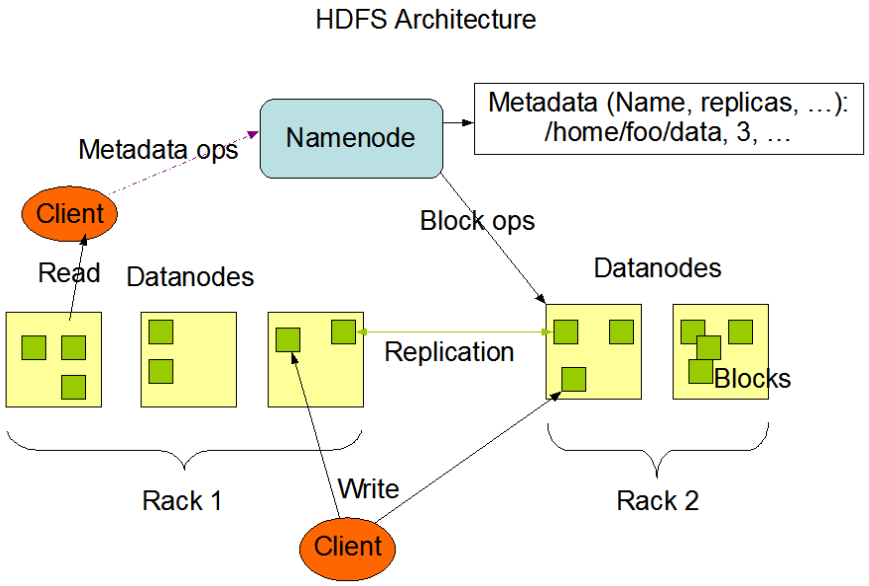


Figura 17:Arquitectura HDFS (56)

Para garantizar la tolerancia a fallos el namenode detecta si un datanode no está disponible mediante un proceso de *heartbeat*, en caso de que no esté disponible replica los bloques perdidos en otros datanodes.

Por su arquitectura HDFS no funciona bien con accesos aleatorios, de hecho, donde mejor funciona es en lecturas secuenciales y paralelas, siendo esto, una de las bases principales del paradigma de programación MapReduce.

2.4. Conclusiones

Tras el estudio sobre el estado del arte realizado se han llegado a las siguientes conclusiones para el diseño e implementación de Cloudbook:

- La computación paralela es necesaria, es difícil de implementar manualmente, sobre todo en lenguajes imperativos y se ha avanzado en la investigación de paralelizadores automáticos.
- La computación grid es el mejor modelo de computación para obtener mejores capacidades computacionales, abordar la paralelización y distribución en estos sistemas es una cuestión de actualidad como demuestra el creciente uso de Apache Spark o Apache Flink.
- Cloudbook usará un modelo de computación grid abordando el problema de la paralelización y la distribución: Será una plataforma que distribuya y ejecute un programa en distintas máquinas y, de forma transparente al programador, le proporcione las capacidades computacionales del conjunto de máquinas
- Cloudbook permitirá la programación con un lenguaje imperativo, puesto que es el paradigma más utilizado por los desarrolladores y uno de los objetivos del proyecto es hacer la plataforma lo más accesible posible.
- El código se dividirá en base a funciones. Al igual que la anterior conclusión, para hacer más sencilla la programación en Cloudbook, el centro de la distribución y paralelismo serán las funciones, ya que la división en base a estas puede resultar una división más natural, y no los bucles que exigen un conocimiento más avanzado al desarrollador.

- Cloudbook será una plataforma de computación distribuida y paralela, cuyas principales diferencias respecto a otras soluciones son:
    - OpenMP: En Cloudbook también se aplica el paralelismo mediante directivas, con la diferencia de que solo se aplican estas directivas a las funciones, y no a los bucles o tareas, tampoco se indican los ámbitos de los datos de los bloques paralelos.
    - Spark: Cloudbook usa un paradigma de programación imperativa, ya que aprender a usar correctamente el paradigma de programación funcional exige un aprendizaje y se pretende que Cloudbook exija poco o nada a los desarrolladores.
    - BOINC: Cloudbook es menos rígido que BOINC donde todos los nodos realizan las mismas tareas, aunque con distintos datos, en Cloudbook se pretende que los nodos colaboren para realizar una tarea y se permite la comunicación entre los mismos.
    - Pluto: Cloudbook se basa en un modelo más simple que Pluto, que se basa en el modelo poliédrico, y aunque sea una herramienta automática pide al desarrollador opciones como tamaño de bloque o factor de desenrollado que exigen un conocimiento muy explícito al programador que Cloudbook no exige.
    - CUDA: CUDA exige al programador una forma explícita de programar, lo que va en contra del objetivo de accesibilidad de Cloudbook.
    - Charm++: Al igual que con Cuda, Charm exige al programador realizar los programas definiendo los *chares* de una determinada manera, lo que va contra el objetivo de accesibilidad de Cloudbook.
- Los distintos nodos de la red distribuida se comunican mediante paso de mensajes.
- La sincronización de los nodos será automática, siguiendo el mismo orden que el indicado por el programador en su código fuente.
- El sistema de ficheros distribuido se dejará a elección del usuario, ya que no influye en el funcionamiento de la plataforma. En el prototipo desarrollado en este trabajo se

usará el sistema de ficheros NFS, ya que es el que mejor se adapta a los experimentos en una red local y es ampliamente conocido y aceptado.

3. La plataforma Cloudbook

En este capítulo se presenta la plataforma diseñada y desarrollada en este trabajo, cumpliendo los objetivos previstos abordando los problemas de división automática del código, distribución del mismo, exigiendo al programador de la aplicación secuencial pocos cambios o ninguno.

Este capítulo se divide en tres partes, la primera presenta las tecnologías usadas en el desarrollo, la segunda es una descripción de alto nivel de la plataforma, en la que se explican las decisiones de diseño y el funcionamiento de la misma, en la tercera parte se muestra una descripción de bajo nivel de los distintos módulos de la plataforma.

Todos los componentes desarrollados en este proyecto y descritos en este capítulo se encuentran en un repositorio de código abierto:

https://github.com/Cloudbook-project

3.1. Tecnologías usadas para el desarrollo de Cloudbook

3.1.1. Python

El código del proyecto se ha desarrollado utilizando el lenguaje Python, que es asimismo el lenguaje que se ha utilizado como base para los programas que se van a paralelizar. Es un lenguaje de programación interpretado de alto nivel, interactivo, y multiparadigma. Usa tipado dinámico, es multiplataforma, y basa su sintaxis en la indexación, para facilitar la legibilidad del código (57).

Python fue creado en 1991 por Guido van Rossum en el Stichting Mathematisch Centrum en los Países Bajos como sucesor de un lenguaje llamado ABC. Y en 2001 se fundó la Python Software Foundation, una organización sin ánimo de lucro creada específicamente para ser la poseedora de toda la propiedad intelectual relativa a Python. Python se distribuye bajo una licencia de código abierto denominada Python Software Foundation License, compatible con la licencia general GNU (58).

Las principales características del lenguaje Python son:

Multiparadigma: Python no fuerza al desarrollador a programar de una forma determinada si no que admite distintos paradigmas. Admite completamente los paradigmas de programación orientada a objetos y de programación estructurada. Y en menor medida también soporta muchas características de otros paradigmas como programación funcional (lista intensionales y diccionarios), programación orientada a aspectos y mediante extensiones se pueden incorporar más paradigmas.

Zen of Python (59): The zen of Python es un documento que resume y da indicaciones sobre la filosofía de programación Python, en la que prima un código claramente legible, y fácil de entender.

Facilidad de extensión: Python está diseñado para ser fácilmente extensible antes que construir toda la funcionalidad en el núcleo del lenguaje. Es muy fácil construir y usar extensiones en C y C++ que cubran o mejoren funcionalidades de Python, como puede ser la optimización de la velocidad, permitiendo así mantener la claridad del lenguaje en lugar de complicarlo.

Gestión de la memoria: Python optimiza la gestión de la memoria usando tipado dinámico y conteo de referencias para liberar los recursos. Además, usa resolución dinámica de nombres, enlazando nombres de variables y métodos durante la ejecución del programa.

3.1.2. Herramientas

Gracias al diseño de Python es fácil diseñar módulos que amplían su funcionalidad o herramientas que la complementan. Algunas de estas herramientas creadas por la comunidad de Python, y que han sido de utilidad en el desarrollo de Cloudbook, son:

Radon (60) es una herramienta de Python que permite obtener métricas sobre el código fuente de un programa. Las métricas que mide son:

- Complejidad ciclomática de McCabe: Esta medida de complejidad se corresponde con el número de decisiones que contiene un bloque de código, es decir es el número de rutas lineales independientes a través del código.
- Métricas en bruto: Estas métricas incluyen las líneas de código, líneas lógicas de código (declaraciones), comentarios, y ofrece las relaciones entre las mismas métricas

- Métricas de Halstead: Mide el vocabulario de un programa, la duración, la longitud, el volumen, la dificultad, el esfuerzo y una aproximación del tiempo para programarlo y número de bugs en base al número de operadores y operandos distintos y totales.
- Índice de mantenibilidad (Visual Studio): Esta métrica mide la facilidad de mantenibilidad de un software. Se calcula en base a las líneas de código la complejidad ciclomática y el volumen de Halstead.

Flask (61) Es un microframework web escrito en Python clasificado de esta manera porque no requiere herramientas o librerías particulares para funcionar. No contiene características que sí contienen otros frameworks, como puede ser una capa integrada de compatibilidad con bases de datos, pero tiene soporte para extensiones que le permiten aplicar la funcionalidad de las que carece de forma transparente, como validación de formularios, manejo de carga, interacción con bases de datos o herramientas de autenticación.

La principal ventaja de su uso en Cloudbook es que una gestión multihilos eficiente y al estar en Python puede cargar sin problemas el código final distribuido.

- **Python Lex Yacc (PLY)** (62) es una implementación de las herramientas Lex y Yacc, es decir las herramientas de análisis léxico y gramático, implementada en Python. Proporciona la mayoría de las funciones estándar de lex/yacc, como la compatibilidad de gramáticas ambiguas, recuperación de errores, y generación de reglas de precedencia. Realiza análisis sintáctico ascendente LR para gramáticas libres de contexto, por su eficiencia y adecuación para gramáticas más grandes. En el proyecto se ha decidido usar esta herramienta en lugar del módulo *parser* que incorpora Python por los siguientes motivos:No se va a realizar un análisis exhaustivo sobre el lenguaje Python, solo sobre las funciones, estructuras de control y otras estructuras que se pueden considerar de alto nivel, y en este caso, no es necesario analizar todos los tipos de tokens que produce Python, que incluyen un alto nivel de recursividad.
- PLY permite usar una estructura para los tokens a la que se puede añadir y modificar la información que necesitemos.

3.2. Descripción de alto nivel

3.2.1. Principios de diseño

En este apartado se describen los principios seguidos en el desarrollo de la plataforma de cara a su implementación.

Lo primero que hay que tener en cuenta es que en este proyecto se resuelven dos problemas distintos, el primero es el referido a la división del código fuente de forma automática, y el segundo es el problema de la distribución del código dividido en el primer problema y las comunicaciones entre los distintos nodos que esto conlleva.

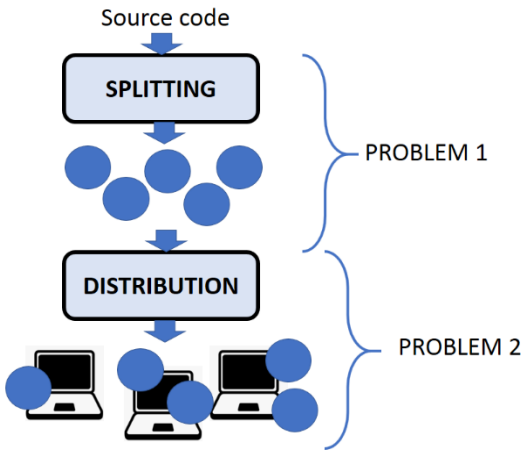


Figura 18: Los dos problemas de Cloudbook

3.2.1.1. Problema 1: División de código

Para resolver este problema es necesario realizar un análisis estático del código, para el que hay que tener en cuenta, en primer lugar, cuál es la unidad mínima que consideramos para la distribución, que en Cloudbook es la función. La plataforma Cloudbook distribuye y paraleliza a nivel de función y no de bucle o por dependencia de datos. Esto se hace porque se realiza la distribución en base a las relaciones entre las funciones. El análisis estático del código se realiza usando la herramienta de Python PLY

La información necesaria para la ejecución de cada módulo que venga de un módulo anterior, o tenga que traspasarse al módulo siguiente, se encapsula en una estructura diccionario de Python, de forma que toda la información recabada en el análisis y división de código fuente esté disponible en el último paso.

Los ficheros de configuración necesarios para iniciar el proceso y los ficheros que se produzcan estarán escritos en formato JSON y en todos los módulos habrá una función auxiliar para leer la información de estos ficheros y convertirla en una estructura diccionario de Python que sea compatible con la información que usan los módulos.

Los ficheros de configuración necesarios, tanto como el código deben estar en una ruta bien conocida que es en la que se ubicará el sistema de ficheros distribuido, esta ruta será:

- En Windows: $HOMEDRIVE/$HOMEPATH/Cloudbook/
- En Linux: /etc/Cloudbook/

Para indicar paralelismo a nivel de funciones se usarán directivas o pragmas, con un formato bien definido: __CLOUDBOOK:PRAGMA__

Hay características del código fuente original que implican la generación de código para permitir la distribución, como el uso de variables globales o de directivas.

3.2.1.2. Problema 2: Distribución de código

Los ficheros de código distribuido se guardan en el sistema de ficheros distribuido, en la ruta indicada en el apartado anterior. De esta forma todos los nodos del círculo de máquinas pueden acceder al código que tienen que ejecutar.

Por otra parte, las comunicaciones entre funciones distribuidas se realizan mediante el protocolo HTTP, de la siguiente manera. Las invocaciones en el código distribuido son de la forma:

invoker([unidad desplegable], función, parámetros)

Que se interpreta como una llamada GET con la unidad desplegable y el nombre de la función.:

http://host/invoke?invoked_function=unidad_desplegable.función

Los parámetros de la función se encuentran en el *body* de la llamada.

3.2.2. Arquitectura global

En este apartado se muestra la arquitectura de la plataforma Cloudbook con la que se abordan los dos problemas de la automatización de la distribución y paralelización: La división del código fuente y la distribución del código dividido

Cloudbook se ha diseñado para funcionar en lo que se ha definido como *modo local*, para mostrar un prototipo sin problemas de disponibilidad y controlable en un laboratorio. Pero está diseñado para poder extenderlo al *modo servicio*, a continuación, se describen estos dos modos.

Modo local (Cloudbook as a tool): En este modo Cloudbook se usa como un conjunto de comandos que se invocan desde la consola, Este modo está destinado al uso privado, en un laboratorio o en una red local, donde las máquinas son fácilmente identificables. Esto se útil para el desarrollador. Este modo usa los siguientes comandos:

Comando	Descripción
Make	Genera las unidades desplegables en un sistema de ficheros distribuido.
Deploy	Genera el diccionario de unidades desplegables y máquinas donde se encuentran.
Run	Invoca la función main en la unidad desplegable 0.
Gui	Interfaz gráfica del agente desde la que se lanza el mismo.

Tabla 1: Comandos Modo Local

Modo servicio: Este modo está destinado al uso público o privado, usando máquinas fuera de la red local y círculos públicos. Este modo de funcionamiento requiere un conjunto de servicios para asignar un agente a un círculo, para invocar al *maker*, realizar el despliegue y la ejecución. Se puede considerar el modo servicio como una extensión del modo local, en el que la funcionalidad está provista como un servicio en la nube al igual que la gestión de círculos, es decir, los comandos del modo local son reemplazados en el modo servicio por servicios en la nube.

En ambos modos, las partes esenciales de Cloudbook son:

- El software: El programa original.
- Cloudbook: El servicio encargado de dividir el programa en unidades desplegables. Este servicio está compuesto de distintos módulos como el analizador de grafos o el

splitter, complementado con otros componentes como el *deployer* o el "gestor de círculos".

- Los agentes: Son los programas encargados de ejecutar el código de las unidades desplegables en cada nodo.
- Círculo de máquinas: Conjunto de nodos que ejecutan un programa.

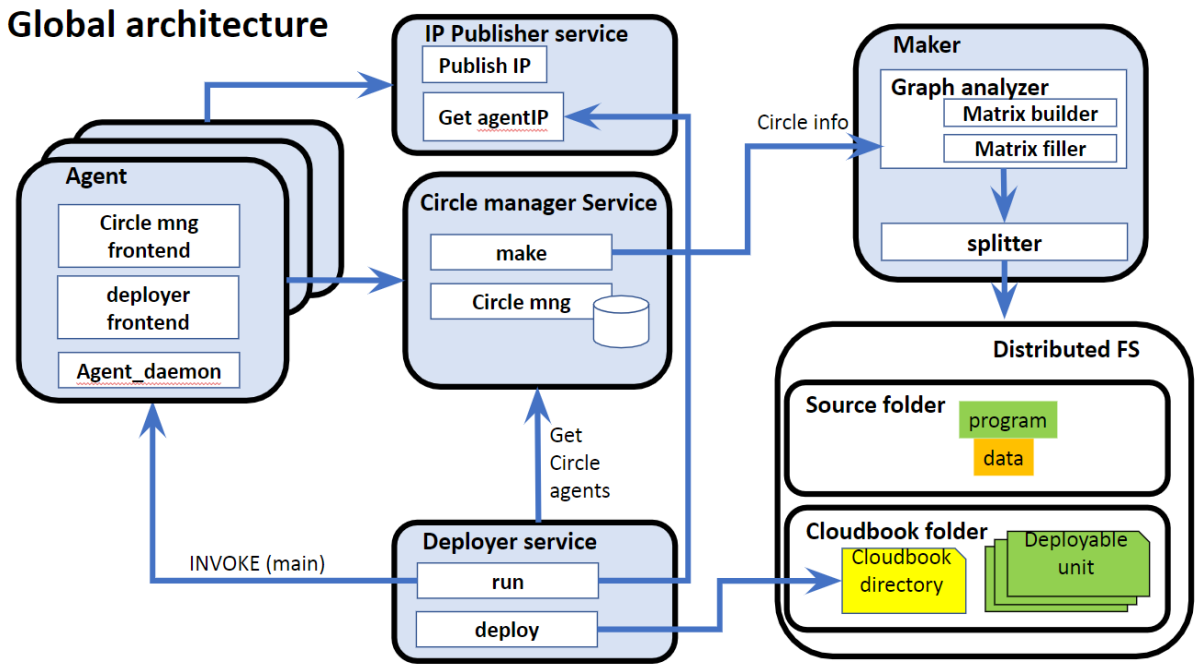


Figura 19: Arquitectura de Cloudbook

Descripción de los componentes de Cloudbook:

- **Agente**: El componente que habrá en cada máquina que forme parte de un círculo. Tareas:
  - Modo local: Ejecutar código e interactuar con otros agentes.
  - Modo local: Indicar IP, e Id de agente en fichero para *local ip publisher*.
  - Modo servicio: Interacción con el servicio de gestión de círculos.
  - Modo servicio: Indicar su IP privada o pública, su Id de agente y de círculo al servicio *Ip publisher*.
- **Servicio de gestión de círculos**: Este componente permite la creación y edición de círculos de máquinas en los que se ejecutaran los programas distribuidos. La información del círculo es necesaria para ejecutar el *Maker*. Tareas:

- Creación de círculos e incorporación de máquinas a los mismos.

En modo local este servicio no es necesario, ya que todas las máquinas forman parte del círculo local.

- **Maker**: Este componente recibe el código fuente y realiza:
  - Análisis del grafo: Analiza y parsea el código para estudiar el grafo de invocaciones representado como la matriz de invocaciones.
  - División del programa: Agrupa funciones basándose en distintos criterios, en piezas de código que son las unidades desplegables, cuyo número depende de la definición del círculo que contiene el número de agentes y máquinas.
- **Sistema de ficheros distribuido**: Este módulo almacena el código y los datos, es accesible por todos los agentes.

  El sistema de ficheros distribuido no condiciona a los agentes. Todas las máquinas del círculo montan el sistema de ficheros distribuido como un directorio local y lo usan de esa manera.

  Por defecto la ruta del sistema de ficheros distribuidos será:

  - En Windows: $HOMEDRIVE/$HOMEPATH/Cloudbook/
  - En Linux: /etc/Cloudbook/
- **Deployer**: Este módulo es el responsable de crear el diccionario *cloudbook* que contiene la asignación de las distintas unidades desplegables a los agentes y también permite empezar la ejecución. Tareas:
  - Crear el diccionario *cloudbook*
  - Cuando se invoca el comando “run”, empieza la ejecución
- **IP Publisher:** El módulo de publicación de direcciones IP. Este módulo recibe información de los agentes, su id, y su IP pública y las guarda. Tareas:
  - Mantiene IDs e IPs de cada agente actualizada.
  - Cuando es requerido por uno de los agentes o el *deployer*, indica qué agentes están actualizados y ejecutándose.
  - Local Ip Publisher: En modo local las IPs se guardan en un fichero en el sistema de ficheros distribuidos. No es un servicio, esta tarea se realiza en el agente.

3.2.3. Estrategia de compilación

El proceso principal de Cloudbook es la *trans-compilación* del código original al código preparado para la distribución y su ejecución en los agentes.

En este apartado se describen las distintas partes implicadas en este proceso.

3.2.3.1. Parseo de código fuente

Cloudbook realiza un análisis léxico del código fuente original para extraer las funciones y las relaciones entre las mismas.

Los tokens que analiza son:

['IMPORT','FUN_DEF','COMMENT','LOOP_FOR','LOOP_WHILE','IF','ELSE',

'TRY','EXCEPT','PRINTV2','PRINTV3','FUN_INVOCATION',

'PYTHON_INVOCATION','INVOCATION','ASSIGNATION','RETURN','IDEN',

'GLOBAL','PARALLEL','RECURSIVE']

Que se almacenan en una tupla de la forma:

<(Nombre Token, valor, línea, indentación)>

- Nombre Token: Nombre definido para el token.
- Valor: El valor del token varía en función del tipo de token:
    - Para estructuras de control y bucles: Es una aproximación del número de veces que se va a ejecutar
    - Para funciones: Es el nombre jerárquico de la función, con el módulo o submódulos a los que pertenece.
    - Para el resto de los tokens es el valor de la línea tras quitar los espacios en blanco.
- Línea: El número de línea donde se encuentra el token.
- Indentación: El nivel de indentación en el que se encuentra la línea, en Python la indentación es especialmente importante por lo que se ha de tener en cuenta. Si hay varios tokens en una línea, se iguala la indentación para todos.

3.2.3.2. Directivas

Las directivas implementadas en el prototipo son:

- Para funciones paralelas: La directiva es *#__CLOUDBOOK:PARALLEL__* las funciones que tienen esta directiva son no bloqueantes por lo que no se permite que devuelvan nada.

  La ejecución de la función se realiza como una sección crítica. Estas funciones se despliegan en todas las unidades desplegables, y se invoca cada vez a una distinta. Cuando son invocadas su funcionamiento consiste en lanzar un hilo con la función y devolver inmediatamente el control a la función invocadora.

  En caso de querer sincronizar los hilos y esperar a que todos terminen, se usará otra directiva que se describe en este apartado.

- Para funciones recursivas: La directiva es *#__CLOUDBOOK:RECURSIVE__* su comportamiento está diseñado para maximizar el nivel de recursividad. Las funciones que tienen esta directiva son bloqueantes.

  Estas funciones se despliegan en todas las unidades desplegables, y se invoca cada vez a una distinta.

- Para funciones locales se usa *#__CLOUDBOOK:LOCAL__* : Las funciones locales son funciones que solamente se usan junto con otra, por defecto Cloudbook las unirá, pero esta directiva permite que las funciones locales a funciones paralelas se desplieguen en todas las unidades desplegables junto con la función paralela con la que interactúan y así evitar cuellos de botella. Todas las invocaciones a estas funciones son locales dentro de la unidad desplegable, no producen tráfico.

- Para sincronizar funciones paralelas se usa *#__CLOUDBOOK:SYNC__*: Esta etiqueta se usa para esperar a que todos los hilos lanzados por una función paralela terminen. Este mecanismo de espera se escribe en el punto en el que se pone la directiva.

3.2.3.3. Variables globales

El uso de variables globales se gestiona mediante la siguiente estrategia:

- Cada variable global se convierte en una función de gestión de la misma. Esta función sólo se despliega en una unidad desplegable, e implementa un mecanismo de cerrojo

para gestionar las operaciones concurrentes ya que las operaciones sobre las mismas son secciones críticas.

- El valor de la variable global se gestiona como un atributo interno que se mantiene durante la ejecución del programa.
- La función de gestión de la variable global tiene un atributo interno para gestionar la versión de la variable global, cualquier cambio sobre el valor de la misma cambia también la versión.
- Cada función que usa la variable global pide su última versión al declararla, y la guarda como un atributo interno de la función que es el usado durante la ejecución.
- Si se quiere "refrescar" el valor de una variable global, hay que declararla de nuevo con la palabra reservada *global* y el nombre de la variable. No hace falta usar una directiva de Cloudbook para esto.

3.2.3.4. Unidades desplegables en los agentes

El código de los agentes está preparado para cargar una unidad desplegable y realizar las comunicaciones necesarias con otras unidades en las funciones que estén en otro agente. Esto se produce de la siguiente manera:

- En las unidades desplegables se crea un objeto llamado *invoker* que no significa nada por sí mismo.
- Las invocaciones a funciones dentro de la unidad desplegable son de la forma:

  invoker(lista_unidades_desplegables, función, parámetros)
- En el agente que carga la unidad desplegable asigna el objeto *invoker* a una función propia que evalúa la función si es local, o construye el mensaje para comunicarse con otro agente que implementa la función solicitada.

3.3. Descripción de bajo nivel del prototipo

3.3.1. Módulo maker

El módulo *maker* es el encargado de analizar el código fuente original para generar las unidades desplegables que se distribuirán en una red de ordenadores. Este módulo agrupa los

submódulos *graph analyzer* y *splitter*, encargados de realizar el grafo de llamadas de las funciones del programa, procesarlo y producir los nuevos ficheros de código fuente.

Es el componente que prepara el código del usuario para poder ejecutarse en Cloudbook, se usa con el comando *make*.

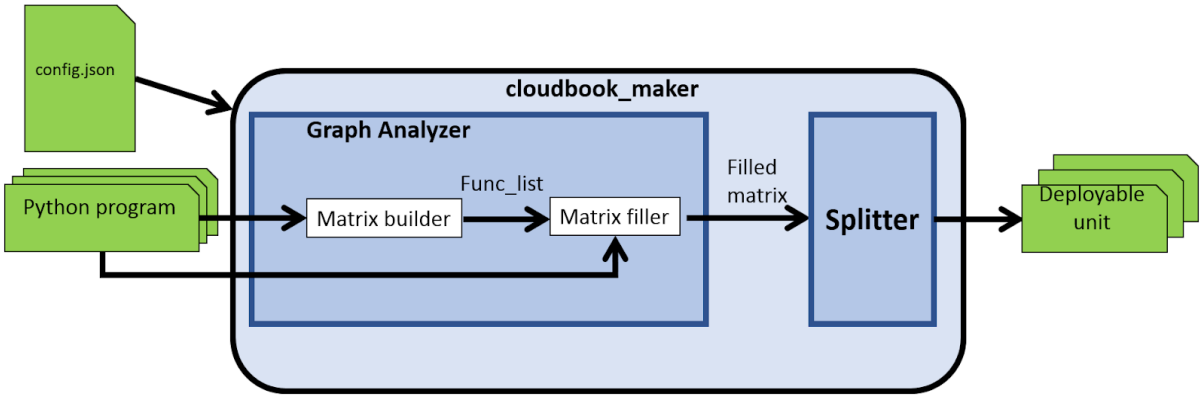


Figura 20: Diagrama módulo maker

Las entradas de este módulo son:

- El fichero *config.json*: De este fichero se obtiene el número deseado de unidades desplegables.
- El código original: El código fuente escrito por el usuario.

Las salidas de este módulo son:

- Unidades desplegables: Ficheros de código en Python con el código que se ejecuta en Cloudbook. este código es una adaptación del código fuente original. Los nombres de estos ficheros son *du_x.py* donde x es un número que tienen asignado.
- du_list.json: Este fichero contiene una aproximación del coste de las unidades desplegables. Se produce una vez se han generado las unidades desplegables usando la herramienta *radon* de Python.

El fichero config.json contiene la siguiente información:

```
{
   "circle_info":{
         "NAME": Nombre del círculo,
         "CIRCLE_ID": Id del círculo local,
```

"DESCRIPTION": Descripción,

"NUM_ATTACHED_AGENTS": N,

"DISTRIBUTED_FS": Ubicación por defecto,

"STATUS": "IDLE|EXECUTING",

"MAX_THREADS": M

}

}

El fichero du_list.json contiene la siguiente información:

{

"du_0": {"cost": Coste, "size": Tamaño en líneas},

"du_N": {"cost":XX ,"size":YY }

}

3.3.1.1. Graph analyzer

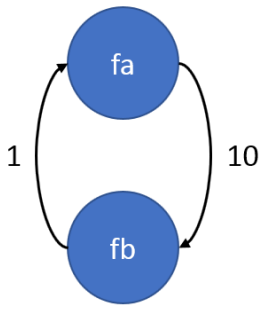
Este componente es el encargado de realizar un análisis estático del código y generar un grafo representado por una matriz de las invocaciones del programa. El grafo de invocaciones es un grafo valorado dirigido G = (V, A) donde:

- V es el conjunto de funciones del programa
- A es el conjunto de arcos valorados por el número de veces que invoca el primer miembro del arco al segundo.

Por ejemplo: Sea un programa con dos funciones, fa y fb, y que fa invoca 10 veces a fb, y a su vez fb invoca a fa 1 vez. El grafo G= (V, A) se define como:

- V = {fa,fb}
- A = {(fa,fb),(fb,fa)} donde (fa,fb) tiene peso = 10 y (fb,fa) tiene peso = 1

El grafo G y su matriz asociada se representan como:



M	fa	fb
fa	0	1
fb	10	0

Figura 21: Ejemplo de grafo de invocaciones

Matrix Builder

Este componente crea el grafo de invocaciones, pero no calcula los pesos de los arcos. En este componente se analiza el código fuente original y se genera una lista de funciones y una matriz vacía que se rellena en el siguiente paso.

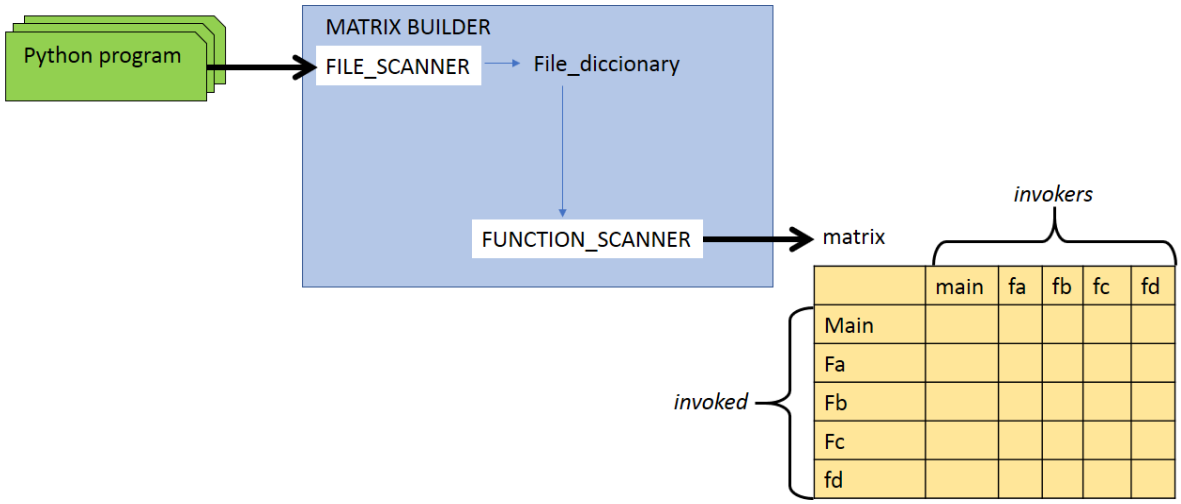


Figura 22:Diagrama Matrix Builder

Este módulo se compone de:

- **File Scanner**: Este componente se encarga de generar un diccionario con todas las carpetas y ficheros del directorio de trabajo. Su objetivo es determinar qué ficheros de código fuente hay que analizar para obtener el grafo de invocaciones.

  Analiza la estructura de ficheros del código fuente y detecta que ficheros de código fuente hay que analizar. Devuelve un diccionario de carpetas y ficheros en formato JSON

Figura 23: Funcionamiento file scanner

- **Function Scanner**: Este componente se encarga de parsear los ficheros de entrada y encontrar las funciones con nombre del programa. Hay dos casos especiales en este paso:
    - **Variables globales:** En este paso se deben detectar las variables globales del programa, ya que en Cloudbook se tratarán como funciones.
    - **Directivas de Cloudbook:** En este paso las directivas opcionales de Cloudbook se guardan en un diccionario de Python para tenerse en cuenta en los siguientes pasos.

  Con la información recabada en el *file scanner*, el *function scanner* realiza un análisis léxico para obtener los nombres de las funciones definidas en el código, utilizando expresiones regulares y el módulo PLY de Python. Los tokens necesarios para obtener los nombres de funciones y sus expresiones regulares (simplificadas) son:

  - FUN_DEF: (def) + (iden) + (\() + (iden) + (\)) + (:) + (\n)*
  - GLOBAL: ^(iden) | (global) + iden + (\n) *

  Para las directivas los tokens y expresiones regulares son:

  - PARALLEL: (#__CLOUDBOOK:PARALLEL__)
  - RECURSIVE: (#__CLOUDBOOK:RECURSIVE__)

  Tras este proceso se crea el diccionario de directivas *labels_dict* y la lista de funciones con la que se representa la matriz vacía.

Matrix Filler

Este componente es el encargado de valorar el grafo de invocaciones y por lo tanto rellenar la matriz, realizando un segundo análisis sobre el código fuente original.

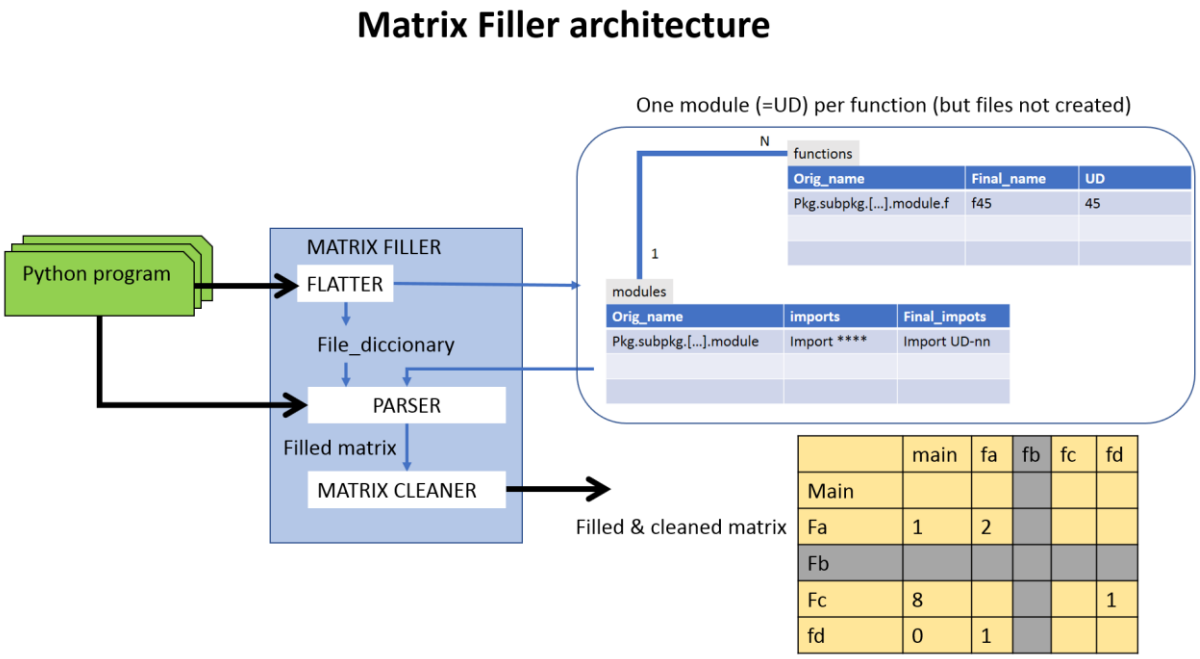


Figura 24: Diagrama Matrix Filler

Este submódulo está compuesto por:

- **Flatter**: Este componente se encarga de "aplanar" el programa, es decir, descompone el programa a nivel de función, la estructura en módulos y submódulos del código fuente se descompone en funciones que están todas al mismo nivel.

  Esto se realiza de forma lógica, no genera todavía ningún fichero. También se traduce los nombres de las funciones del programa a nombres tipo *fx* siendo x su índice en la lista de funciones, así se evitan problemas en el código final si dos funciones de dos módulos distintos tienen el mismo nombre. De la misma forma se le asigna un nombre de unidad desplegable *du_x*. Y se guarda en la tabla FUNCTIONS.

  Con los imports del programa original realiza la misma tarea, los traduce por el nombre de unidad desplegable si son imports internos al programa, y los deja igual si son librerías de Python. Esta información la guarda en la tabla MODULES.

  Para obtener los imports del programa se usa de nuevo el módulo PLY y expresiones regulares:

  - IMPORT: (from) + (iden) + (import) + (iden) | (import) + (iden)

  En las tablas FUNCTIONS y MODULES está toda la información necesaria para rehacer la estructura del programa.

Por último, hay que observar un caso especial, la función main del programa, que actúa como punto de entrada al mismo, siempre será asignada la du_0 y la f0.

- **Parser**: Analiza las funciones para obtener información de la interacción entre las mismas y el número de invocaciones o una estimación del mismo.

  Para ello, se vuelve a analizar el código original para ver qué invocaciones hay en cada función, teniendo en cuenta que las declaraciones de una variable global en una función son invocaciones a la variable global, que se trata como una función.

  Se asigna un valor a cada invocación en función de en qué parte del código se encuentre:

  - Si está dentro de un bucle se intenta evaluar el número de iteraciones, si no se puede se asigna automáticamente un 100.
  - Si está dentro de otras estructuras de control, como *if, else, try* o *except* se asigna un 1, sin evaluar si entra en la estructura o no.
  - Si no está dentro de ninguna estructura, se le asigna 1.
  - Los valores que se asignan a una misma invocación se multiplican entre sí, si están anidados, si no, se suman.

  Usando de nuevo el módulo PLY y expresiones regulares en el parser analizamos las siguientes:

  - INVOCATION: (iden) + (\() + (iden) + (\)) + (\n)*
  - GLOBAL: ^(iden) | (global) + iden + (\n) *
  - LOOP_FOR: (for) + (target_list) + (in) + (expression_list) + (\:)
  - LOOP_WHILE: (while) + (expression) + (\:)
  - IF: (if) + (expression) + (\:)
  - ELSE: (else) + (\:)
  - TRY: (try) + (\:)
  - EXCEPT: (except) + (\:)

- **Matrix Cleaner**: Módulo auxiliar que elimina el código de las funciones que no participan en la ejecución del programa, así reduce la matriz, y el tamaño del código que se va a desplegar.

Diseño iterativo del módulo graph analyzer

Como se ha explicado en la metodología de trabajo en la sección tres del capítulo uno, se ha realizado un diseño incremental de los distintos módulos hasta llegar a la versión del prototipo presentado en este trabajo. En el graph analyzer se han realizado las siguientes versiones:

- Versión 0: En esta versión se validan las bases sobre las que se construyen las siguientes.
    - Capacidad de aplanar código.
    - Generar la matriz rellena.
    - Solo permite funciones sin parámetros.
- Prototipo: En esta versión se valida funcionalidad más avanzada que permiten probar casos de uso.
    - Permite funciones con parámetros.
    - Permite uso de variables globales.

3.3.1.2. Splitter

Este componente se encarga del desarrollo de las unidades desplegables, esto es, analiza el grafo de invocaciones y en función de un criterio dado unifica las distintas funciones y genera los ficheros de código que se distribuirán en las distintas computadoras que forman parte del círculo en el que se ejecute el programa distribuido.

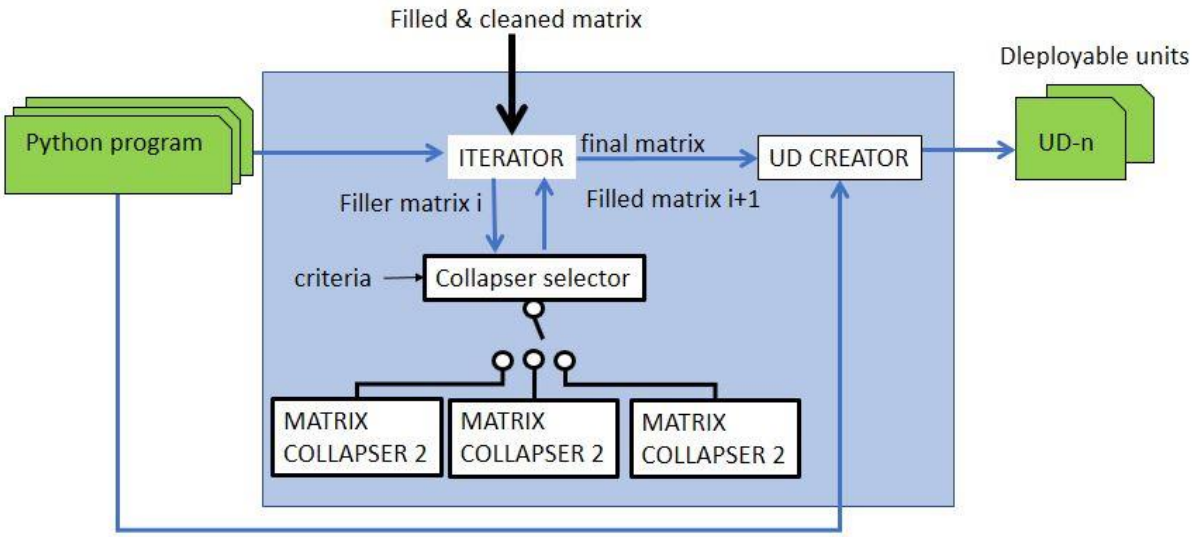


Figura 25: Diagrama Splitter

Este componente incluye:

- **Iterator**: En función de la matriz con todas las funciones y el número de unidades desplegables que se quieren implementar se eligen de dos en dos, funciones que se van a juntar en un mismo fichero.

  En cada iteración hasta obtener el número de unidades desplegables deseadas, el *iterator* elige un criterio que devuelve las funciones de la matriz que se deben colapsar y las colapsa produciendo una nueva matriz más pequeña.

  El proceso termina cuando la matriz tiene tantas filas y columnas como unidades desplegables deseadas.

- **Collapser selector**: Elige qué criterio se seguirá para colapsar la matriz de invocaciones, cada criterio analiza la matriz y devuelve la fila y columna que deben colapsar (unirse). Los criterios actuales son los siguientes:
  - Low Latency (ll): Colapsa siempre las funciones que tienen más comunicación entre ellas, para evitar tráfico externo que ralentice el programa.
  - Low Latency Cost Size Balance (llcsb): Colapsa las funciones que más se comunican entre ellas, pero balancea el número de funciones en cada unidad desplegable.

Si cuatro funciones se llaman mucho entre ellas, se irían las cuatro a la misma unidad desplegable y el resto pueden quedar con pocas funciones. Con este criterio, se podrían separar de dos en dos en distintas unidades desplegables.

- **Collapser**: Este módulo se encarga de colapsar (unir) las funciones indicadas por el *collapser selector* y actualizar los nombres de unidad desplegable en la tabla FUNCTIONS, ya que inicialmente cada función en una unidad desplegable, y los imports necesarios para su ejecución en la tabla MODULES.

  También hay que observar un caso especial, siempre que una función se colapse con la función main, el conjunto de funciones quedará representado como la unidad desplegable cero (du_0) que es la que iniciará la ejecución en el módulo deployer.
- **Du Creator**: Este componente genera los ficheros ".py" con el código a ejecutar por cada computador basándose en lo indicado finalmente por la matriz colapsada y el código fuente con los nombres traducidos por los que serán usados.

  Antes de empezar a generar los ficheros de unidades desplegables, se añaden a la matriz colapsada las funciones necesarias según las etiquetas puestas por el usuario. Si el usuario ha marcada una función como *parallel* se añade en todas las filas y columnas de la matriz para que el *du_creator* las tenga en cuenta.

  Este módulo, escribe un fichero de código para cada grupo de funciones, por lo que consulta el código original y escribe la unidad desplegable nueva. Sus principales tareas son:

  - Unificar los distintos imports de los ficheros de las funciones que forman parte de la unidad desplegable.
  - Crea el objeto invoker, necesario para la comunicación entre los agentes.
  - Busca cada función en el código original y la copia en la unidad desplegable teniendo en cuenta los tipos de función:
    - Funciones normales, traduce el nombre de función, usando *fx* en lugar del nombre original.
    - Funciones marcadas *parallel*: En este caso se escriben dos funciones, la primera, que se escribe como *fx* y es la encargada de lanzar un hilo con la ejecución de la función y responder inmediatamente al invocador.

```
def fx(parameter1, parameterN):
    threadfx = threading.Thread(target= parallel_fx, daemon = False, args = [parameter1, parameterN])
    threadfx.start()
    return json.dumps("thread launched")
```

Tabla 2:Codigo para lanzar hilos en función paralela

- La función que lanzada como hilo que realiza la tarea como una sección crítica se llama *parallel_fx*
- La declaración inicial de variables globales: Se traduce como una función con 3 atributos propios que son:
  - El valor de la variable global
  - El número de versión
  - Un cerrojo para proteger las operaciones concurrentes.

La función tiene dos parámetros, la operación que se va a realizar y el número de versión de la invocadora. Si la operación es "None" se devuelve el valor de la variable global si el número de versión es menor que el actual, o se devuelve "None" si el invocador tiene la versión más actualizada, en cualquier caso, no se ejecuta ninguna operación sobre la variable.

Si se realiza una operación sobre la variable, se coge el cerrojo para impedir accesos concurrentes y se ejecuta aumentando el número de versión.

- La declaración de variables globales dentro de una función: Añade a la función 2 atributos propios:
  - El valor local de la variable global
  - El número de versión local de la variable global

En la declaración de la variable global siempre se solicita la versión más nueva de la misma, mediante una llamada con código de operación "None".

- Busca dentro de cada función en el código original y la copia en la unidad desplegable traduciendo las invocaciones teniendo en cuenta:
  - Las invocaciones se traducen de la forma:

        - fun(variables) => invoker ([du_x],fx,variables)
    - La traducción de nombres, se usan los nombres de tipo *fx* en lugar de los nombres originales.
    - Las declaraciones de variables globales se traducen como llamadas a la función asociada a la variable global para obtener la última versión.
    - Las operaciones sobre variables globales se traducen por operaciones que se ejecutan en la función que gestiona la variable global.
    - Las invocaciones a funciones marcadas como *parallel* y *recursive* están asignadas a códigos de unidad desplegable especiales para que los agentes llamen a una unidad desplegable distinta en cada operación.
    - Si encontramos la directiva *sync* incluimos el código necesario para la sincronización de hilos:

```
while json.loads(Cloudbook_th_counter("")) > 0: #This was sync
                    time.sleep(0.01)
```

Tabla 3: Código para gestionar sincronización

La función *Cloudbook_thread_counter(value)* es una función que genera el du_creator para gestionar las funciones *parallel* es una función que implementa un mecanismo tipo semáforo para esperar a que terminen todas las funciones lanzadas. Tiene dos atributos propios, el valor, y el cerrojo, para protegerse de las llamadas concurrentes. Y con cada llamada con valor "++" suma uno el valor, y cuando las funciones paralelas terminan llaman con "--" restando uno al valor. Una llamada sin valor devuelve el valor actual para saber cuántos hilos vivos hay.

```
def Cloudbook_th_counter(value):
  if not hasattr(Cloudbook_th_counter, "val"):
        Cloudbook_th_counter.val = 0
  if not hasattr(Cloudbook_th_counter, "cerrojo"):
        Cloudbook_th_counter.cerrojo = Lock()
  if value == "++":
```

```
            with Cloudbook_th_counter.cerrojo:
                Cloudbook_th_counter.val += 1
    if value == "--":
            with Cloudbook_th_counter.cerrojo:
                Cloudbook_th_counter.val -= 1
    return json.dumps(Cloudbook_th_counter.val)
```

Tabla 4: Código de control de hilos

Diseño iterativo del módulo splitter

Como se ha explicado en la metodología de trabajo en la sección tres del capítulo uno, se ha realizado un diseño incremental de los distintos módulos hasta llegar a la versión del prototipo presentado en este trabajo. En el splitter se han realizado las siguientes versiones:

- Versión 0: En esta versión se validan las bases sobre las que se construyen las siguientes.
    - Validar la división de código con las siguientes restricciones:
        - Sin objetos.
        - Sin variables globales.
        - Sin directivas.
        - Sin parámetros en las funciones.
- Prototipo: En esta versión se valida funcionalidad más avanzada que permite probar nuevos casos de uso.
    - Permite funciones con parámetros.
    - Permite uso de variables globales.
    - Permite el uso de directivas.

3.3.1.3. Bases de datos usadas en el proceso

Durante el proceso *make* se guarda en una base de datos SQLite dos tablas con información sobre las funciones del programa, y su estructura, las tablas FUNCTIONS y MODULES.

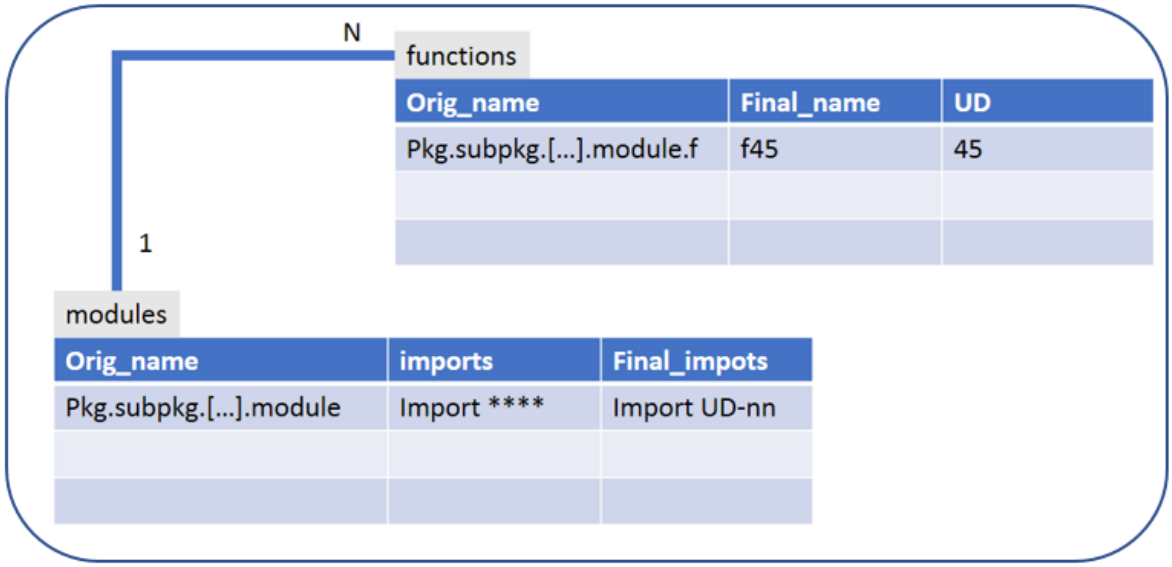


Figura 26: Bases de datos en proceso Make

Durante el proceso de análisis del código, es necesario recoger información sobre los ficheros del programa para redistribuir su contenido. La información que se guarda es:

- Nombre original y jerárquico de cada función.
- Nombres traducidos de las funciones, que serán los que usen las unidades desplegables, para evitar problemas con funciones de mismo nombre en distintas ubicaciones.
- Las importaciones de cada fichero.
- Traducciones de las importaciones de cada fichero con el nombre de unidad desplegable que necesita importar cada función.

3.3.2. Deployer

Este es el módulo encargado de guiar el despliegue y ejecución de los programas. Se encarga de crear el fichero *Cloudbook.json* que contiene la información necesaria para distribuir las unidades desplegables producidas por el módulo *maker* en las distintas máquinas y de iniciar la ejecución del mismo. Realiza estas tareas mediante dos comandos:

- Deploy: Este comando ejecuta la creación del fichero *Cloudbook.json* que contiene información para asignar unidades desplegables a los agentes. Este fichero se guarda en el sistema de ficheros distribuido para que los agentes puedan acceder a él en cualquier momento.

- Run: Este comando comienza la ejecución del programa realizando la llamada inicial al punto de entrada de la unidad desplegable 0.

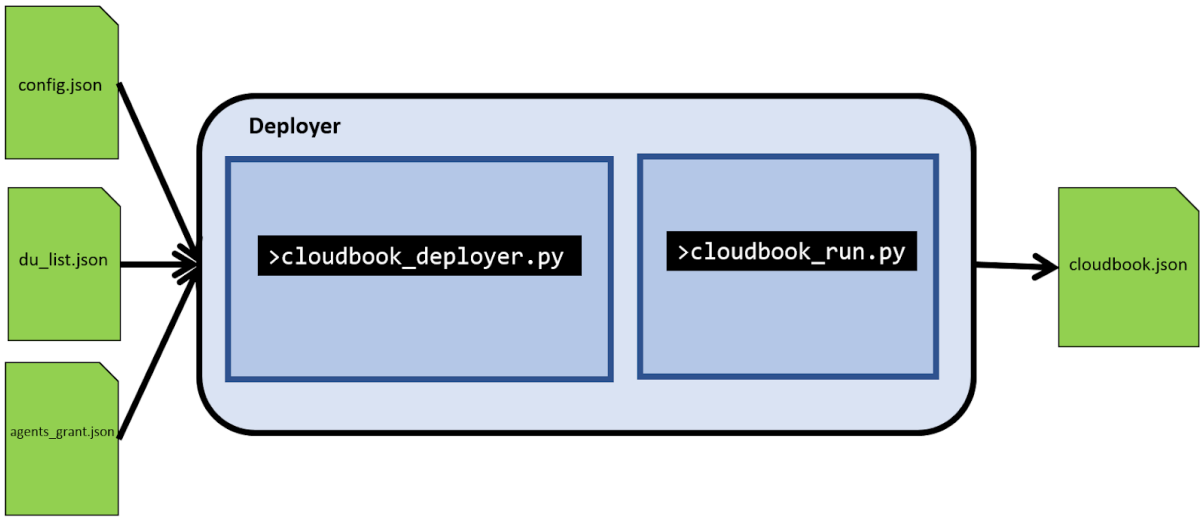


Figura 27: Diagrama módulo deployer

La entrada de este módulo son los siguientes ficheros:

- El fichero *agents_grant.json*: Este fichero es producido por los agentes y en el apuntan la identificación de cada agente y una estimación cualitativa de su potencia proporcionada por el dueño de la computadora donde se ejecuta el agente.
- El fichero *du_list.json*: Este fichero es producido por el módulo *maker* y contiene una estimación de coste de las unidades desplegables.
- El fichero *config.json*: El fichero de configuración global de Cloudbook en el que está la ruta del sistema de ficheros distribuido, que permite cambiar la ubicación por defecto.

La salida de este módulo es un fichero:

- Deploy: La salida del submódulo *deploy* es el fichero *Cloudbook.json* que contiene la asignación de agentes a las distintas unidades desplegables.

El formato del fichero *Cloudbook.json* es:

{

‘du_0’: Lista de agentes que lo despliegan,

‘du_N’: Lista de agentes que lo despliegan

}

El módulo *deployer* se ejecuta a calculando la mejor asignación de unidades desplegables a agentes teniendo en cuenta los costes y potencia de cada uno y escribiendo esta asignación en el fichero *Cloudbook.json* para que los agentes sepan que unidad desplegable tienen que cargar.

Hay un caso especial en la asignación, y es respecto al agente 0, que es el que se despliega en la máquina que va a iniciar la ejecución. En este agente tiene que estar la unidad desplegable 0 que es la que contiene el punto de entrada del programa.

Una vez se ha creado el fichero *Cloudbook.json* se puede ejecutar el comando *run*, que realiza una llamada a la función *main* de la unidad desplegable cero, la llamada que realiza es la siguiente: http://localhost:3000/invoke?invoked_function=du_0.main

Una vez realizada la petición, se inicia la ejecución y se queda esperando la respuesta con el resultado de la ejecución.

Diseño iterativo del módulo deployer

Como se ha explicado en la metodología de trabajo en la sección tres del capítulo uno, se ha realizado un diseño incremental de los distintos módulos hasta llegar a la versión del prototipo presentado en este trabajo. En el deployer se han realizado las siguientes versiones:

- Prototipo: Este módulo tiene una funcionalidad más reducida, por lo que las hipótesis validadas desde la primera versión han servido para el funcionamiento del prototipo:
    - Produce el fichero *Cloudbook.json* a partir de los ficheros *du_list.json* y *grant_agentes.json*
    - Funciona en modo local.

3.3.3. Agente

Este módulo es el encargado de ejecutar el código generado en el módulo *maker* y cuya distribución se ha indicado en el módulo *deployer*. En este apartado se muestra el agente en modo local, que se diferencia del agente en modo servicio, en que no tiene interacción con el gestor de círculos, y usa el módulo *local ip publisher*.

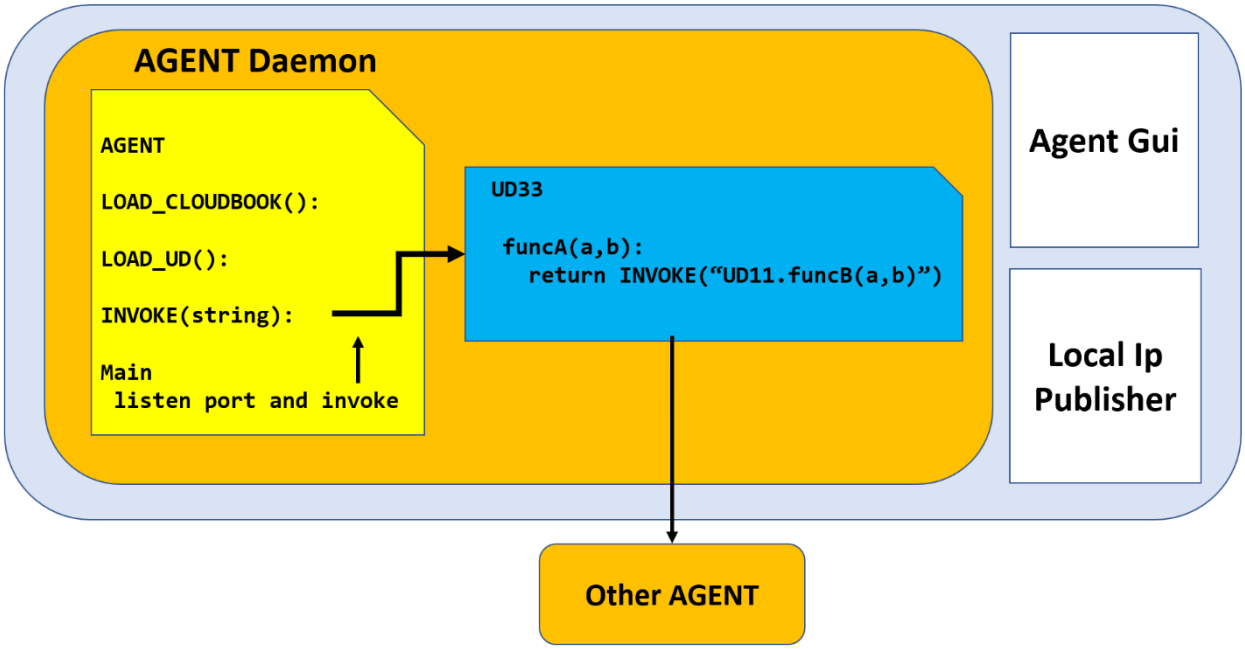


Figura 28: Diagrama agente

Todos los agentes son iguales, pero en el proceso de despliegue se fuerza a que el agente 0, que es el que contiene el punto de entrada del programa sea el que se queda en la computadora que ha realizado todo el proceso, *make*, *deploy*, y arrancar los agentes.

3.3.3.1. Necesidad de ejecución multihilo

Una vez iniciada la ejecución, todos los agentes se comunican entre sí y se pueden producir interbloqueos a nivel de función, por ejemplo, una unidad desplegable 0 invoca una función a en la unidad desplegable 1, y durante la ejecución de la función a, tiene que realizar un cálculo mediante la función b que es independiente y está en la unidad desplegable 0, como la unidad desplegable 0 está esperando el resultado de la unidad desplegable 1, no puede atender esa petición y se quedaría bloqueada, por eso es necesario que abra un hilo para ejecutar la función b, y comunicar su resultado a la unidad desplegable 1 que los está esperando [Figura 29], mientras tanto la unidad desplegable 0 no se queda esperando el resultado de a, sino que sigue realizando tareas.

Es por esto que los agentes deben ser multihilo, para atender llamadas a otras funciones que estén en el mismo agente.

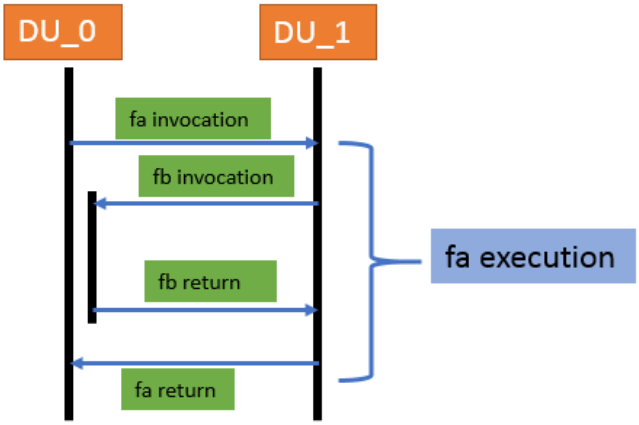


Figura 29:Ejecución multihilo en los agentes

3.3.3.2. Protocolo de comunicación entre agentes

El protocolo usado para la comunicación de los agentes es HTTP debido a la versatilidad delmismo y fácil implementación que permite realizar y validar cualquier test de funcionamiento. La principal ventaja es que se pueden implementar fácilmente en Python con el framework Flask que permite una gestión multihilos eficiente y al estar en Python puede cargar sin problemas las unidades desplegables. La principal desventaja es el rendimiento comparado con otros protocolos de capa de aplicación orientados al paso de mensajes, o protocolos sobre la capa de transporte vía UDP o TCP.

3.3.3.3. Componentes del agente

En modo local el agente tiene tres componentes:

- **Gui**: Este submódulo es la interfaz del agente que facilita la creación y administración de los distintos agentes que se encuentran ubicados en la misma máquina. Tiene las siguientes funcionalidades:
    - Por defecto la interfaz muestra los agentes “instalados” en la máquina, es decir los que se han creado usando la interfaz y guardan sus atributos en el fichero *confi_id_agent.json*. Muestra el estado de los agentes, que puede ser *stop* u *online*, el círculo al que pertenecen, y la estimación cualitativa que se proporciona al crear el agente.

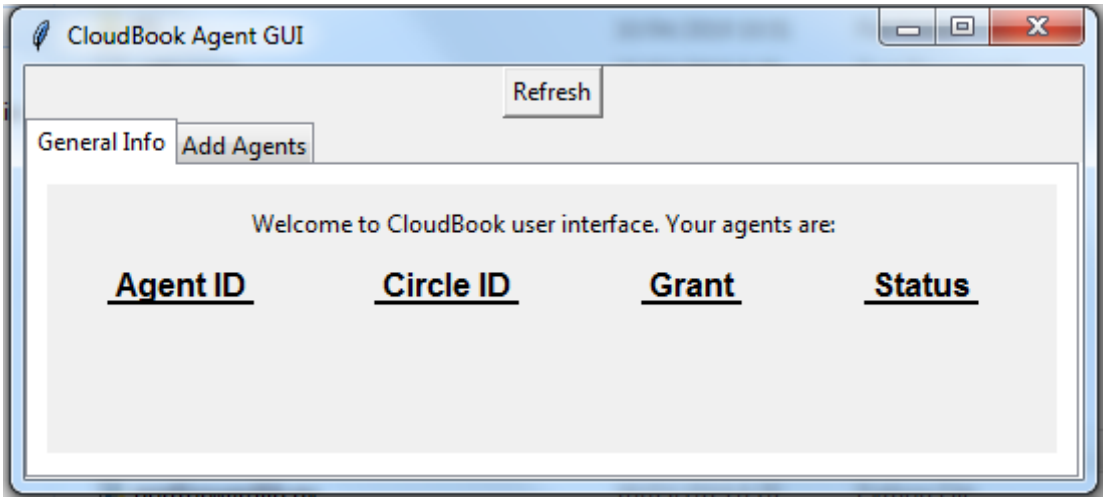


Figura 30: Interfaz inicial agente

- Pestaña para añadir agentes: En esta pestaña se muestra el formulario para añadir un agente, el círculo en el modo local esta predefinido, a continuación, se incluye la aproximación cualitativa (*low*, *médium*, *high*) de la potencia que proporciona el agente al círculo y la localización del sistema de ficheros distribuido.

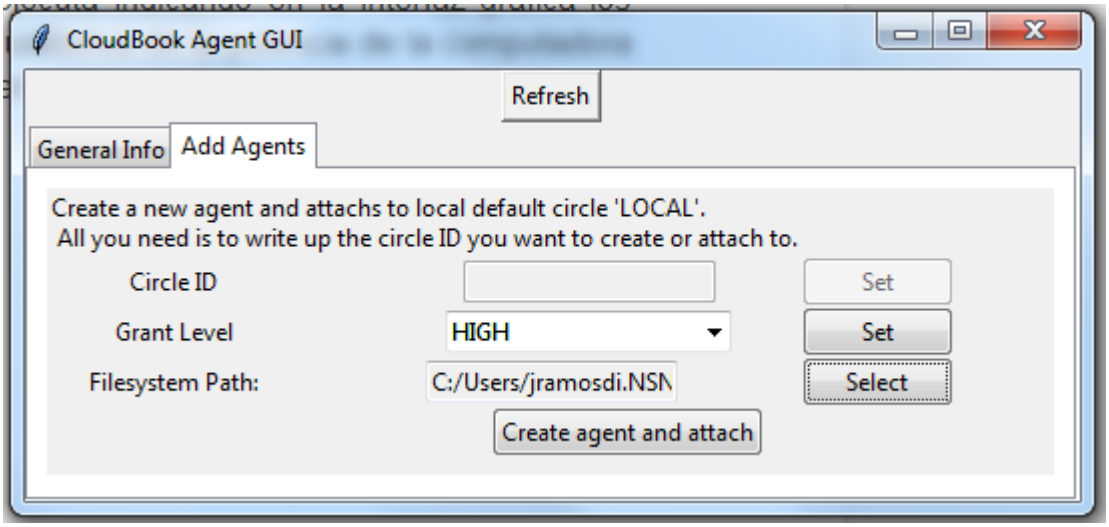


Figura 31: Pestaña de creación de agentes

En esta operación se genera el fichero *agents_grant.json* que contiene la información sobre la aproximación de potencia que usará el módulo *deployer*.

- Launch: Arranca el agente, espera a que se generen el fichero *Cloudbook.json* y las unidades desplegables para poder cargarlas. También actualiza o genera

el fichero *local_ip_info.json* que contiene las direcciones IP de los agentes que pertenecen al círculo.

- Execution: Después de arrancar el agente y que estén los ficheros necesarios cargados, el agente responde las distintas peticiones de los agentes, y la petición inicial del programa realizada por el módulo *deployer* con el comando *run*.
- Stop: Este botón paraliza la ejecución del agente y mata cualquier proceso asociado a esa ejecución. El agente puede ser relanzado luego.

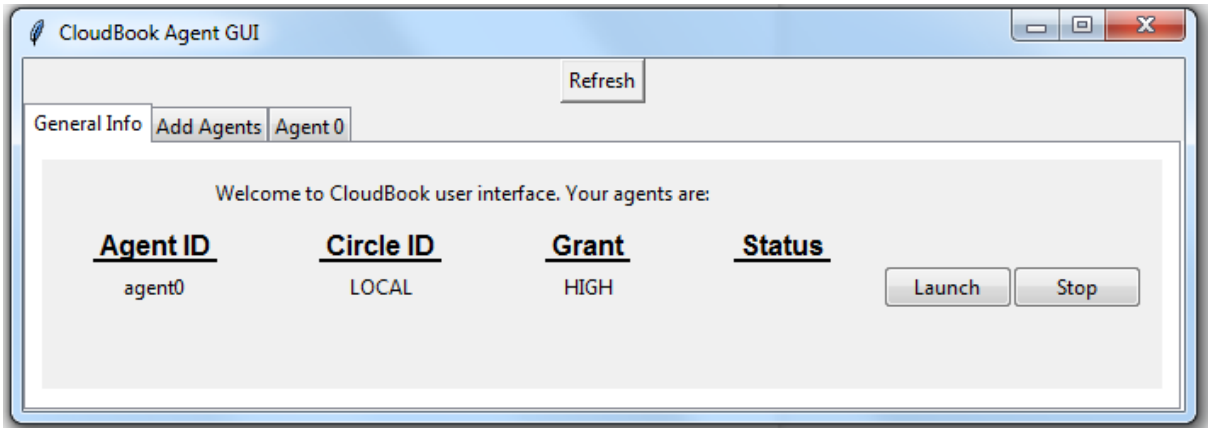


Figura 32: Interfaz con agente preparado para ejecutarse

- **Agent *Daemon***: El agente gestiona la ejecución de las diferentes funciones que se han incluido en las unidades desplegables. Primero cuando se ejecuta desde el botón "Launch" de la GUI el agente anuncia su IP en el fichero *local_ip_info.json*, entonces carga las unidades desplegables asignadas. Luego, cuando el *deployer* inicia la ejecución, ejecuta la función *main*() (si la contiene) o espera a que otro agente llame a una función que contenga. Los diferentes agentes tienen la capacidad de, al ejecutar código de su propiedad, llamar a un agente diferente que contiene la funcionalidad necesaria mediante peticiones basadas en HTTP. Entonces, el agente tiene dos tipos de funcionalidades:
  - Funcionalidad para crear y editar agentes: estas funciones se invocan desde la GUI y están relacionadas con las funciones de creación y edición de agentes mencionadas. Internamente, estas funciones crean o modifican los diferentes ficheros de configuración si es necesario, como la propia configuración del

fichero del agente, el *agents_grant.json* si el parámetro modificado es la concesión del agente, etc.

- Funcionalidad para la ejecución de agentes: esto podría considerarse como el propio agente. Contiene diferentes funciones para gestionar la invocación de funciones y llamadas para el resto de los agentes. Al ejecutar una función, la invocación puede considerarse como "local" o "remota". Con la invocación local, el agente sabe que debe la función, por lo que la ejecuta directamente. Es un poco más complicado cuando el agente no posee la función invocada, estas funciones se han traducido por llamadas a la función *invoker*. En este caso, el agente comprobará cuál de los agentes ejecuta la unidad desplegable que necesita consultando el archivo *Cloudbook.json*. Después, comprobará el *local_ip_info.json* para obtener la dirección IP de ese agente. Finalmente, construirá la petición http y le pedirá a ese agente que ejecute la función. Dependiendo del tipo de función, el agente de llamadas debe esperar una respuesta o no.

- **Local ip Publisher**: Su principal tarea es descubrir y almacenar la dirección IP del agente en un archivo ubicado en el sistema de ficheros distribuido. Tiene funciones para descubrir la dirección IP de la máquina y comprueba cuál es la de la red local, desechando las otras que puedan estar relacionadas con localhost o con software para máquinas virtuales.

El módulo agente produce los siguientes ficheros:

- config_id_agent.json: Contiene la información propia del agente, se crea cuando se añade un agente en la GUI. Y tiene el siguiente formato:

```
{
   "AGENT_ID": Identificación del agente,
   "DISTRIBUTED_FS": Ruta predefinida al sistema de ficheros distribuido,
    "CIRCLE_ID": "LOCAL",
    "GRANT_LEVEL": High, medium o low
}
```

- local_ip_info.json: En este fichero se guardan las direcciones IP de los distintos agentes. Tiene el siguiente formato:

{

identificación de agente1: {"IP": ip del agente},

identificación de agenteN: {"IP": ip del agente}

}

- agents_grant.json: En este fichero se guarda la relación entre agentes y capacidad proporcionada. Tiene el siguiente formato:

{

identificación de agente1: High, medium o low,

identificación de agenteN: High, medium o low

}

Diseño iterativo del módulo agente

Como se ha explicado en la metodología de trabajo en la sección tres del capítulo uno, se ha realizado un diseño incremental de los distintos módulos hasta llegar a la versión del prototipo presentado en este trabajo. En el agente se han realizado las siguientes versiones:

- Versión 0: En esta versión se validan las bases sobre las que se construyen las siguientes.
    - Se comunican funciones sin parámetros
    - No hay interfaz gráfica
    - No hay soporte para directivas
- Prototipo: En esta versión se valida funcionalidads más avanzada que permite probar nuevos casos de uso.
    - Permite funciones con parámetros.
    - Tiene interfaz gráfica para facilitar su uso.
    - Soporta el uso de directivas.

3.3.4. Sistema de ficheros distribuido

En este apartado se trata el sistema de ficheros distribuido que usa la plataforma Cloudbook, aunque este elemento es intercambiable, se describe a continuación la estructura que tiene que tener, y en la segunda parte de este apartado se detalla la implementación de NFS como sistema de ficheros distribuido que es el recomendado para usar Cloudbook en modo local debido a que es sencillo de implementar y funcionará en un entorno local donde es fácil de controlar.

El filesystem distribuido tiene la siguiente estructura y ubicación:

- El directorio raíz estará en:
    - En Windows: $HOMEDRIVE/$HOMEPATH/Cloudbook/
    - En Linux: /etc/Cloudbook/
- La estructura de la carpeta Cloudbook será:
    - Cloudbook/config/: Con los ficheros:
        - config.json de configuración global
        - config_agent_id_agent.json : Uno por agente, con la configuración del mismo.
    - Cloudbook/original: Con el código fuente original
    - Cloudbook/distributed:
        - du_list,json: Lista de unidades desplegables con aproximación del coste.
        - agents_grant.json : Lista de agentes con aproximación cualitativa de la potencia.
        - local_IP_info.json: Lista de agentes con su ip local.
        - Cloudbook.json : Asignación de agentes a unidades desplegables.
        - Cloudbook/distributed/du_files/:
            - du_x.py: Los distintos ficheros de unidad desplegable.

Para el modo local se recomienda usar NFS, aunque se puede usar cualquier otro.

4. Experimentación y resultados

En este capítulo se analiza el prototipo presentado con distintos programas, para validar el funcionamiento de la división y distribución automática del código, y por otro lado analizar los resultados que ofrece en ejecución en un banco de pruebas determinado.

Este capítulo se divide en dos partes, en la primera se presentan los programas que se van a validar, que son dos pruebas de concepto para validar el prototipo, y el tercero un caso de uso real en el que se ha usado el prototipo. En la segunda parte, se presenta la configuración de los entornos donde se han ejecutado las pruebas, y se analizan los resultados de la ejecución de los programas, tanto de la división y distribución de código como de los resultados de la ejecución en las máquinas distribuidas frente a la ejecución secuencial para la que los programas fueron diseñados.

El código asociado a las pruebas de concepto de este capítulo se encuentra en:

https://github.com/Cloudbook-project/Cloudbook_examples

4.1. Pruebas de concepto y casos de uso

4.1.1. Prueba de concepto: N-Body por fuerza bruta

El problema de los n cuerpos consiste en determinar las distintas interacciones de un grupo de partículas de acuerdo a la ley de gravitación universal de Newton.

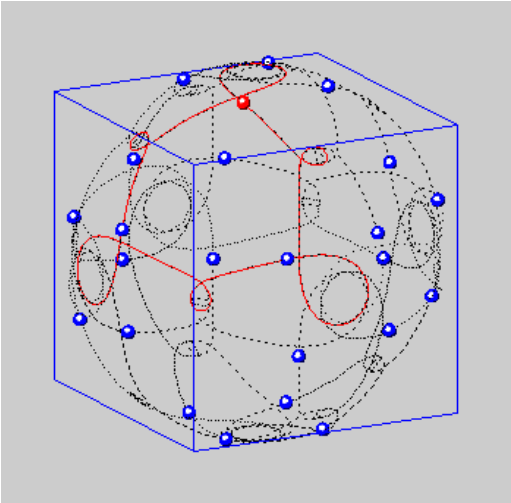


Figura 33: Diagrama NBody (63)

Los motivos para elegir este problema como prueba de concepto son:

- Es un problema con un gran coste computacional, muy difícil de paralelizar, ya que el paralelismo de este problema es de grano fino e implica muchas comunicaciones.
- Permite validar características de Cloudbook como:
    - El paso de variables de gran tamaño entre agentes.
    - Uso de variables globales y su gestión como sección crítica
    - Uso de distintas directivas de Cloudbook para obtener paralelismo.

El coste del algoritmo de fuerza bruta es de O(n^2), para reducir el coste se usan otros métodos para aproximar el resultado, pero para validar el prototipo de Cloudbook basta con el algoritmo de fuerza bruta.

La implementación consiste en:

El algoritmo para calcular nbody por fuerza bruta usa dos variables globales, que son dos conjuntos de cuerpos, es decir, el de entrada y el de salida, que se intercambian una vez se han realizado todos los cálculos, estas variables son "*body_list"* y *"body_new"* respectivamente.

Cada cuerpo es una tupla de 5 elementos, posición x e y (se ha hecho en dos dimensiones), masa y velocidad en ambas dimensiones. La implementación secuencial consta de las siguientes funciones:

- **main**: Esta función tiene dos partes, en la primera ejecuta un bucle según el número de iteraciones (instantes que queremos simular) en la que por cada cuerpo en la variable global body_list llama a la función compute_body, una vez se han calculado todos los cuerpos, se intercambian body_list y body_new para preparar la siguiente iteración. La función devuelve el último body_list calculado, que representa el estado de los cuerpos tras todas las iteraciones.

    Incorpora la etiqueta *__CLOUDBOOK:SYNC__* después de cada cálculo de una iteración, para sincronizar todas las ejecuciones de la función *compute_body* que estará etiquetada como paralela.

- **compute_body**(single_body,iteration): Esta función es la que realiza el cálculo de las fuerzas de un cuerpo respecto del conjunto de cuerpos en un instante dado.

  Tiene un bucle en el que calcula las fuerzas sobre el single_body llamando a la función "compute_contribution_force" respecto a todos los cuerpos del conjunto. Una vez calcula las fuerzas, calcula la aceleración, velocidad y posición y por último añade este cuerpo a la lista de cuerpos nueva, que es la del siguiente instante.

  Esta función contará con la directiva __*CLOUDBOOK:PARALLEL*__ por lo que se distribuirá en todas las unidades desplegables, y cada vez que se invoque, se invocará a una distinta que levantará un hilo en el agente destino para que ejecute la función y retornará rápidamente el control al agente invocador, para que pueda seguir invocando a otros agentes, permitiendo que la función se ejecute en paralelo en todas las unidades desplegables.

- **compute_contribution_force**(bodyA, bodyB): Esta función realiza los cálculos necesarios para obtener las nuevas fuerzas que actúan en los cuerpos.

  Incorpora la directiva ___*CLOUDBOOK:LOCAL*__ por lo que se distribuirá en todas las unidades desplegables como una función local de las mismas. Esto es porque es una función que solo se comunica con *compute_body* que es paralela, y debe ejecutarse de forma local a la misma, para no ralentizar al programa teniendo que realizar una invocación externa en cada cuerpo que se calcule.

4.1.2. Prueba de concepto: Torres de Hanói

El problema de las torres de Hanói consiste en transportar discos apilados de mayor a menor tamaño de un poste a otro, pudiendo pasar por un poste auxiliar. Para mover los discos hay que seguir unas normas:

1. Solo se mueve un disco cada vez.
2. Un disco no puede estar sobre otro más pequeño.
3. Solo se puede desplazar el disco que está en la cima del poste.

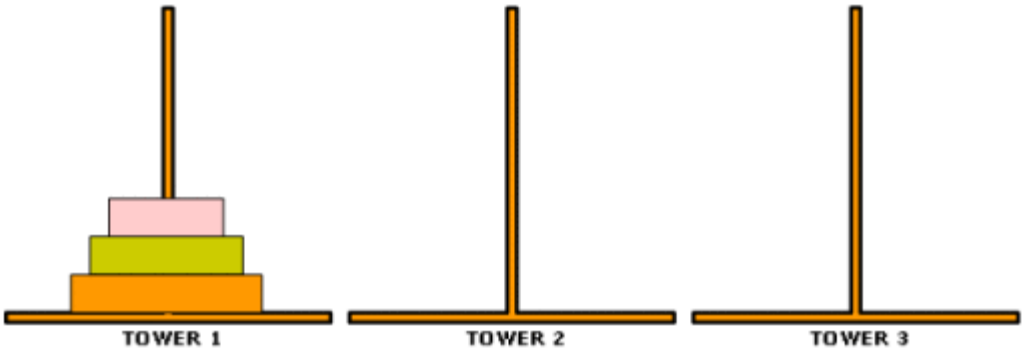


Figura 34: Diagrama Torres de Hanoi (64)

Este es un problema clásico de recursión que se resuelve en 2^n -1 movimientos, siendo n el número de discos total. Como Python no optimiza la recursión de cola, un problema recursivo puede causar un desbordamiento, es por eso por lo que Python tiene un límite de recursión. La motivación para usar este problema como prueba de concepto es validar:

- El primer objetivo de esta prueba de concepto es comprobar que Cloudbook multiplica la profundidad de recursividad de Python al distribuir en distintas máquinas el código.
- El objetivo secundario es validar uno de los objetivos del proyecto, y este es proporcionar una herramienta de distribución y paralelismo por un coste de aprendizaje muy bajo, y permitir que los usuarios no tengan que convertir un algoritmo recursivo en iterativo, porque directamente pueden ejecutar el algoritmo recursivo.

La implementación consiste en:

Escribir 2^n-1 movimientos llevaría demasiado tiempo para el problema con una entrada grande como la que se quiere validar, es por eso que se ha cambiado el algoritmo para que en lugar de darnos los pasos necesarios para resolver el problema nos devuelve el número de pasos realizando las distintas llamadas recursivas. La implementación consta de las siguientes funciones:

- **main()**: Esta función es la que inicia el proceso, contiene la llamada inicial a la función *move_tower*.
- **move_tower(altura)**: Esta función recibe el número de discos de la torre inicial.
    - Si la altura es 1: Devuelve 1, es decir 1 paso es necesario para mover un disco

- Si la altura se mayor de 1: El número de pasos para resolver el problema serán tantos como el doble del resultado de move_tower(n-1) más uno, siendo n el número de pisos.

A esta función se le aplica la etiqueta __*CLOUDBOOK:RECURSIVE*__ que distribuye la función en todas las unidades desplegables para distribuir las llamadas y mantener el orden secuencial de ejecución.

El algoritmo original que produce 2^n -1 pasos se ha implementado, pero por motivos obvios no se puede resolver para números muy grandes puesto que consume demasiado tiempo imprimir todos los pasos. Su implementación es de la forma:

- **main()**: Inicia el proceso con la llamada inicial a *move_tower* con el número de pisos y la identificación de la torre origen, destino e intermedia.
- move_tower(altura, origen, destino, intermedio): Si la altura es mayor que uno:
  - Invoca a *move_tower* con la altura reducida en 1, la torre origen en la posición de origen, la torre intermedia en la posición de destino, y la torre destino en la posición de intermedia.
  - Imprime: Mueve disco de *torre origen* a *torre destino*
  - Invoca a *move_tower* con la altura reducida en 1, la torre intermedia en la posición de origen, la torre destino en la posición de destino, y la torre origen en la posición de intermedia.

A esta función se le aplica la etiqueta __*CLOUDBOOK:RECURSIVE*__ que distribuye la función en todas las unidades desplegables para distribuir las llamadas y mantener el orden secuencial de ejecución.

4.1.3. Caso de uso: Preprocesamiento de flujo de tráfico en training de IDS

Este caso de uso consiste en el análisis de un conjunto de datos con millones de trazas de tráfico real para extraer las características necesarias para poder entrenar un sistema automático de detección de intrusiones (IDS).

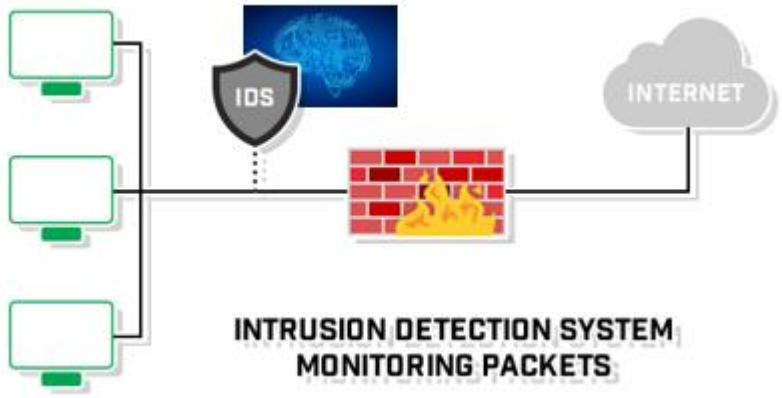


Figura 35: Diagrama IDS (65)

El algoritmo de preprocesamiento elije las características más importantes del conjunto de trazas para convertirlas en una entrada válida para el algoritmo de entrenamiento. Por cada característica elegida, un tipo de procesamiento será requerido para producir la salida, usa una variable global *"done"* para indicar si ya ha procesado una característica.

Este caso de uso permite valorar las ventajas de Cloudbook para distribuir las distintas partes del código de preprocesado y procesar el fichero en paralelo en base a distintas características.

La implementación consiste en:

- **main()**: Esta función controla el flujo de la ejecución del software contando el número de características ya procesadas y asignando nuevas características cuando se pida. Incorpora la etiqueta *__CLOUDBOOK:SYNC__* con el fin de esperar a que todas las características sean procesadas mediante la función *process_piece()* que será etiquetada como paralela. A continuación, indica la creación del archivo final.
- ***assing piece()***: Esta función elige aleatoriamente una característica de las que conforman el conjunto de datos para ser procesada. A esta función no se le aplican etiquetas.
- ***process_piece(piece)***: Esta función recibe el nombre de la característica que tiene que procesar, con este, busca en el archivo del conjunto de datos la característica asignada y realizará el preprocesamiento específico usando distintas librerías de Python para análisis de datos como Scikit-learn o SciPy. Una vez procesados los datos, la característica procesada se guardará en un fichero.

A esta función se le aplica la directiva *__CLOUDBOOK:PARALLEL__* para distribuirla en todas las unidades desplegables y poder procesarla en paralelo con diferentes características.

- ***create_final_dataset()***: Esta función leerá todos los archivos generados por *porcess_piece(piece)* y los unirá en un archivo único con toda la información procesada.

4.2. Resultados

4.2.1. Banco de pruebas

Se han realizado las pruebas en tres entornos controlados.

- Una red local de cuatro Raspberry Pi 3 b+ con las siguientes características:
  - Procesador: Broadcom BCM2837B0, Cortex-A53 64-bitSoC@ 1.4GHz.
  - Ram: 1GB LPDDR2 SDRAM.
  - Wi-Fi + Bluetooth: 2.4GHz y 5GHz IEEE 802.11.b/g/n/ac, Bluetooth.
  - Sistema Operativo: Raspbian

Figura 36: Laboratorio con 4 Raspberrys y HDMI splitter

- Una red local de cuatro instancias de Google Cloud del tipo n1-standard-4 con las siguientes características:

- Procesador: 4vCpus basadas en Intel Xeon Scalable Processor (Skylake) a 2.0 GHz
- Ram:15GB de Memoria
- Sistema Operativo: Ubuntu 18.04

Figura 37: Instancias de Google Cloud

- Un ordenador personal HP EliteBook con las siguientes características:
  - Procesador AMD PRO a10-8700B R6 4Cores @1.8GHz
  - Ram: 8.00 GB
  - Sistema operativo: Windows 7 64bits

Como filesystem distribuido se ha usado NFS, instalado en las distintas máquinas y configurado de la siguiente manera:

- Como NFS tiene arquitectura cliente servidor es necesario elegir una máquina como servidora, lo normal es que sea la misma máquina que ejecuta el *maker* y el *deployer*.
- En la máquina servidora hay que permitir el acceso a los clientes a la carpeta compartida, escribiendo en /etc/exports la siguiente línea por cada cliente:
  - /carpeta_para_compartir clientIP(rw,sync,no_subtree_check)
- A continuación, hay que exportar la carpeta compartida y reiniciar el *kernel* de NFS para que tenga efecto:
  - sudo exportfs -a
  - sudo systemctl restart nfs-kernel-server
- En el cliente hay que montar la carpeta compartida en la ruta en la que se quiera tener, con el siguiente comando:

- sudo mount serverIP:/carpeta_para_compartir /carpeta_compartida

4.2.2. N-Body por fuerza bruta

El caso de uso de Nbody por fuerza bruta se ha usado para validar los conceptos desarrollados en la plataforma, la distribución del código y el uso de etiquetas. En este apartado se van a mostrar los resultados obtenidos al ejecutar el algoritmo para distintos números de cuerpos en distintas implementaciones:

- Secuencial en una máquina
- Distribuido y con etiquetas de paralelización

El objetivo de esta prueba es comprobar la penalización de las comunicaciones, el rendimiento de la plataforma y su adaptabilidad para trabajar con algoritmos secuenciales.

Esta prueba se ha realizado en las cuatro Raspberrys en una red local, puesto que en este entorno se pueden comprobar las características de la ejecución de una tarea de forma distribuida y paralela.

4.2.2.1. Procesado en Cloudbook

En este apartado se muestran los resultados más relevantes de la ejecución una vez ha pasado por el prototipo de Cloudbook. Tras su paso por el *maker* indicando en el *config.json* que se desean 4 unidades desplegables se obtienen los siguientes resultados:

- Graph Analyzer:

La primera matriz sin colapsar refleja las relaciones entre las funciones (invocadores/invocados) según el análisis estático:

	Main	Body_list	Body_new	Compute_body	Compute_contribution_force
Main	0	0	0	0	0
Body-list	101	0	0	1	0
Body_new	100	0	0	1	0
Compute_body	10000	0	0	0	0
Compute-contribution_force	0	0	0	100	0

Tabla 5: Matriz sin procesar NBody

Tras el proceso de *collapse* y aplicar las etiquetas, las matrices se han agrupado en función de su relación:

Para que se entienda mejor, los grupos de funciones se consideran de la siguiente manera:

grupo0: [main,compute_body,compute_contribution_force]

grupo1: [body_list, compute_ body,compute_contribution_force]

grupo2: [body_new, compute_ body,compute_contribution_force]

grupo3: [compute_ body,compute_contribution_force]

Matriz	grupo1	grupo2	grupo3	grupo4
grupo1	10000	0	0	0
grupo2	102	0	0	0
grupo3	101	0	0	0
grupo4	100	0	0	0

Tabla 6: Matriz Final NBody

- Splitter: Produce las cuatro unidades desplegables
    - du_0.py:
        - main
        - compute_body

        - compute_contribution_force
        - Cloudbook_thread_counter: Para gestionar los hilos
    - du_1.py:
        - _VAR_body_list: La función que gestiona la variable global body_list
        - compute_contribution_force
        - compute_body
    - du_2.py:
        - _VAR_body_new: La función que gestiona la variable global body_new
        - compute_contribution_force
        - compute_body
    - du_5.py:
        - compute_contribution_force
        - compute_body

- Deployer: el Cloudbook.json

```
{
"du_0": ["agent0"],
"du_1": ["agent1"],
"du_2": ["agent2"]
"du_5": ["agent3"]
}
```

4.2.2.2. Análisis de la ejecución

Durante la ejecución de Cloudbook se producen las siguientes comunicaciones entre agentes:

- Función main:
    - 3 llamadas a body_list: Una para empezar el programa, y 2 que se repiten a cada iteración.

- 2 llamadas a body_new: Que se repiten en cada iteración.
- N llamadas a la función compute_body de las que cada agente recibe N/4. Siendo N el número de cuerpos y 4 el número de agentes [Figura 38].

- Función compute_body:
  - 1 llamada a body_list por iteración
  - 1 llamada a body_new
  - 1 llamada a Cloudbook_thread_counter
- Función compute contribution force: No realiza llamadas, solo la usa compute_body
- Las funciones de gestión de variables globales no realizan llamadas.

Para hacer el análisis se ha realizado una iteración lo que conlleva N llamadas a la función compute_body, que a su vez hará N llamadas a body_new para actualizar un cuerpo, y N llamadas a Cloudbook_thread_counter (véase Figura 38), las llamadas dentro de compute_body se realizarán de forma paralela, por lo que la penalización en tiempo vendrá por las invocaciones a los distintos agentes para que ejecuten nbody.

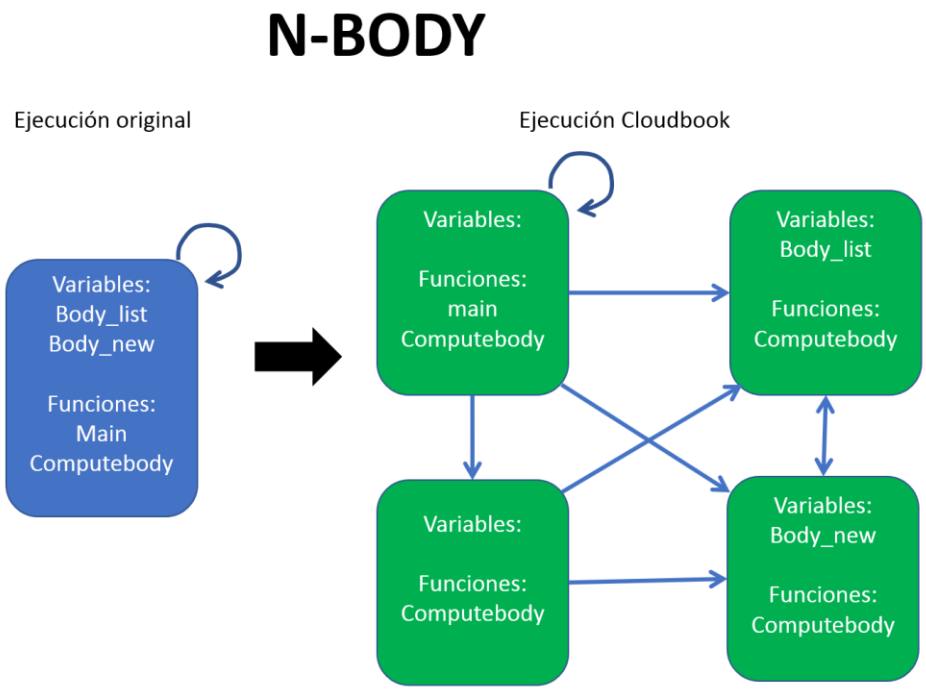


Figura 38: Esquema de ejecución n-body

4.2.2.3. Resultados y discusión

A continuación, se muestran los resultados obtenidos tras la ejecución del caso de uso de forma secuencial y en el entorno de Cloudbook:

cuerpos	secuencial	cloudbook
100	0,17	2,76
500	4,25	19
1000	17,15	43,1
5000	436,5	315
10000	1749,5	935

Tabla 7: Resultados Nbody

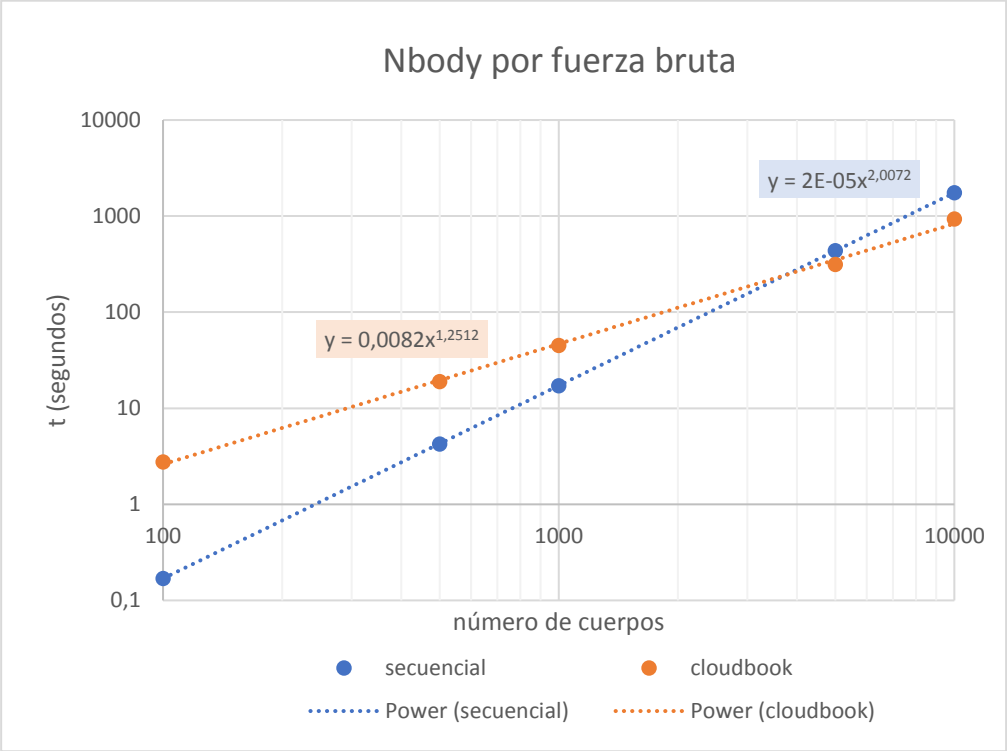


Figura 39: Gráfica resultados NBody

- En cuanto a la ejecución en Cloudbook, esta prueba sirve para validar el funcionamiento del prototipo, como transformar un programa secuencial pensado para ejecutarse en una computadora, y que se transforme automáticamente en un programa distribuido con funciones paralelas.
- En cuanto al rendimiento, como se puede ver por las gráficas los tiempos se produce un cambio cuando el número de cuerpos crece:

- El tiempo del algoritmo en Cloudbook es mayor que el secuencial: Esto ocurre cuando hay pocos cuerpos, se debe a que el cálculo de las fuerzas de un cuerpo respecto al resto (compute_body) que es el que se realiza en paralelo, es mucho más rápido que lo que cuesta invocar a la unidad desplegable y que esta lance un hilo que realice la tarea. Esto implica que cuando le llega la siguiente invocación con otro cuerpo, lleva un tiempo perezoso (en cuanto a la ejecución de compute_body, puede haber hecho otras tareas), pero no aprovecha la paralelización, es más lento que el secuencial, porque hace los cálculos de forma secuencial, parando entre cuerpos porque le tiene que llegar una invocación.
- A medida que el número de cuerpos crece el tiempo del algoritmo en Cloudbook es más rápido que el secuencial, esto se debe a que ya se aprovecha el paralelismo, las tareas (compute_body) tardan más tiempo ya que tienen más cuerpos que calcular, y sí que se realizan cálculos en paralelo. Por ejemplo:
  - Tiempo Invocación: 0,07seg
  - Tiempo compute_body (1000 cuerpos): 0,017seg
  - Tiempo compute_body (1000 cuerpos): 0,17seg

De todas formas, nos alejamos del paralelismo ideal al distribuir una tarea en cuatro máquinas, esto se debe a que, aunque ejecutes en paralelo, se ejecuta de uno en uno, las peticiones llegan para un cuerpo y se tienen que procesar de una en una. La solución para alcanzar el paralelismo ideal sería que la función compute_body, en vez de calcular un solo cuerpo, calculase varios a la vez (ampliar la granularidad del problema), con esto se reducen invocaciones, procesado de comunicaciones y se mejoraría el rendimiento, no de forma ideal porque hay un tiempo de comunicaciones, pero si muy cerca del paralelismo ideal.

Como conclusión a este caso de uso, se puede decir, que los programas con paralelismo de grano fino mejoran la eficiencia si son problemas computacionalmente complejos o con una entrada muy grande. Para un problema sencillo puede no mejorar el tiempo, en el manual de instrucciones se advertirá al programador sobre esto, y se le recomendará agregar cálculos en paralelo, pero esto exige un conocimiento avanzado sobre la naturaleza del programa, que el programador no tiene por qué conocer.

4.2.3. Torres de Hanói

El caso de uso de las torres de Hanói se ha usado para validar el uso de la plataforma con programas recursivos, especialmente el de la etiqueta *__CLOUDBOOK:RECURSIVE__*. El algoritmo que se va a probar es el que produce el número de pasos necesarios para resolver el problema.

En este apartado se van a mostrar los resultados producidos para algoritmos recursivos con una profundidad muy grande. Se van a mostrar distintas implementaciones:

- Secuencial en una máquina.
- Distribuido y con etiquetas de recursividad en dos máquinas.

El objetivo de esta prueba es comprobar el aumento de los recursos computacionales al usar Cloudbook, específicamente el de la pila de llamadas recursivas al tener disponibles las capacidades de dos ordenadores en lugar de uno.

Esta prueba se realiza en el ordenador personal, puesto que el objetivo es comprobar el crecimiento de la pila de llamadas recursivas, se puede comprobar desde dos procesos distintos que serán los dos agentes instalados en una misma máquina.

4.2.3.1. Procesado en Cloudbook

En este apartado se muestran los resultados más relevantes de la ejecución una vez ha pasado por el prototipo de Cloudbook. Tras su paso por el *maker* indicando en el *config.json* que se desean 2 unidades desplegables se obtienen los siguientes resultados:

- Graph Analyzer:

Devuelve la siguiente matriz rellena:

	[main, move_tower]	[move_tower]
[main, move_tower]	0	0
[move_tower]	1	1

Tabla 8: Matriz Torres de Hanói

- **Splitter**: Produce las dos unidades desplegables
    - du_0.py:
        - main
        - move_tower
    - du_1.py:
        - move_tower
- **Deployer**: el Cloudbook.json

```
{
"du_0": ["agent0"],
"du_1": ["agent1"]
}
```

4.2.3.2. Análisis de la ejecución

Durante la ejecución del programa distribuido se van a producir las siguientes comunicaciones:

- Función main: Realiza una llamada a move_tower para iniciar la ejecución del programa.
- Función move_tower: Siendo n el número de discos sobre los que se quiere resolver el problema. Realizará n llamadas a si mismo hasta que n sea 1.

Como move_tower es una función recursiva, cada llamada se producirá a una unidad desplegable distinta, elevando así el número de llamadas recursivas posibles en Python.

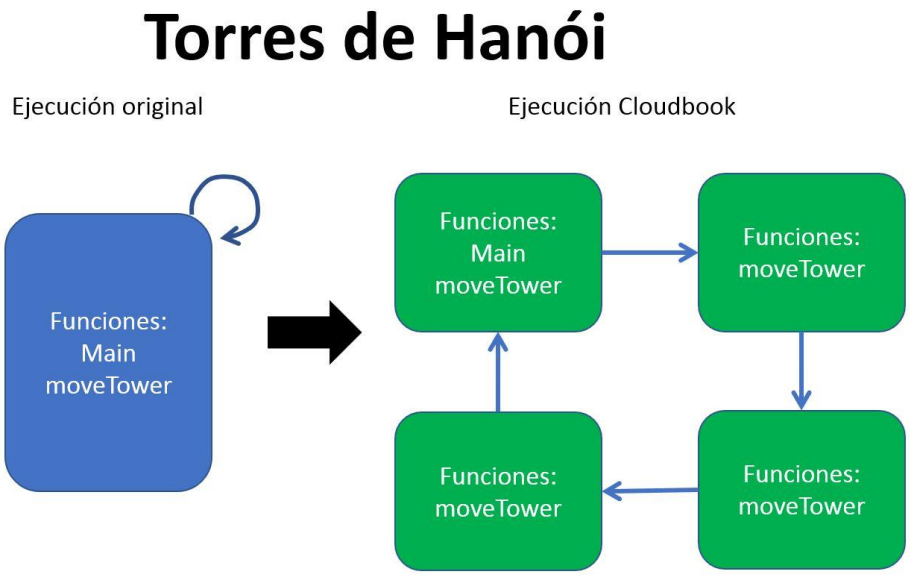


Figura 40: Diagrama funcionamiento Hanoi

4.2.3.3. Resultados y discusión

Como el límite de llamadas recursivas en Python está en mil llamadas, la pila desborda al llegar a mil, como se puede ver en la siguiente Figura.

```
C:\Users\jramosdi.NSN-INTRA\cloudbook\original>py hanoiv2.py
Traceback (most recent call last):
  File "hanoiv2.py", line 16, in <module>
    print(main())
  File "hanoiv2.py", line 12, in main
    result = move_tower(1000,1,2,3)
  File "hanoiv2.py", line 8, in move_tower
    aux = move_tower(height-1,origin,destination,intermediate)
  File "hanoiv2.py", line 8, in move_tower
    aux = move_tower(height-1,origin,destination,intermediate)
  File "hanoiv2.py", line 8, in move_tower
    aux = move_tower(height-1,origin,destination,intermediate)
  [Previous line repeated 993 more times]
  File "hanoiv2.py", line 5, in move_tower
    if height == 1:
RecursionError: maximum recursion depth exceeded in comparison
```

Figura 41: Error en Hanói secuencial (1000 pisos)

Sin embargo, en Cloudbook llega hasta el final porque el tamaño de la pila se ha multiplicado por el número de máquinas.

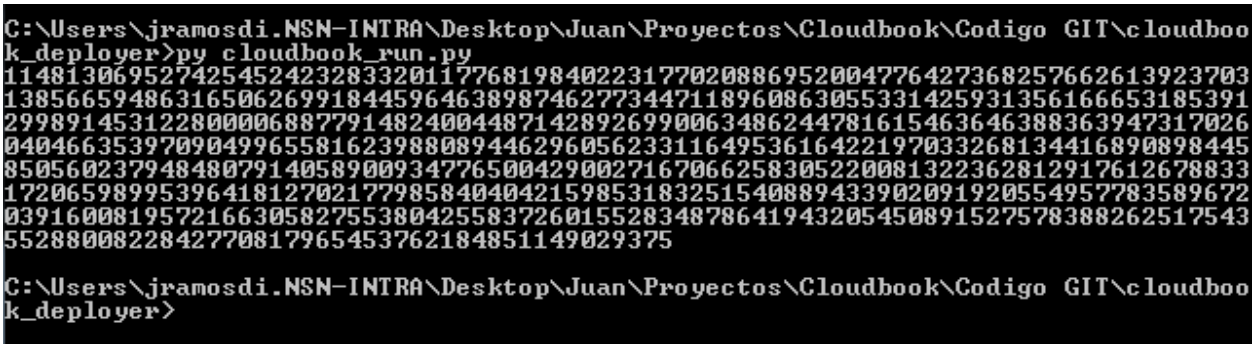


Figura 42: Hanoi 1000 pisos en Cloudbook

Con este resultado validamos que Cloudbook aumenta el tamaño de la pila de llamadas recursivas evitando que se produzca desbordamiento en algoritmos recursivos sin necesidad de convertirlo en iterativo o con recursividad de cola.

4.2.4. Preprocesamiento de flujo de tráfico en training de IDS

Este caso de uso se realiza sobre un problema real, y sirve para validar Cloudbook como herramienta de apoyo en otros trabajos. El problema a tratar involucra paralelismo de grano grueso, ya que se basa en el análisis de un *dataset* muy grande basado en trazas de tráfico en internet.

Debido al alto consumo de memoria de la prueba, se ejecuta en el entorno de Google Cloud que es el que proporciona máquinas distribuidas con mayor capacidad de memoria.

4.2.4.1. Procesado en Cloudbook

En este apartado se muestran los resultados más relevantes de la ejecución una vez ha pasado por el prototipo de Cloudbook. Tras su paso por el *maker* indicando en el *config.json* que se desean 2 unidades desplegables se obtienen los siguientes resultados:

- Graph Analyzer:

La primera matriz sin colapsar refleja las relaciones entre las funciones (invocadores/invocados) según el análisis estático:

	Main	Done	Assign_piece	Process_piece	Create_final_dataset
Main	0	0	0	0	0
Done	1	0	1	0	0
Assign_piece	1	0	0	0	0
Process_piece	1	0	0	0	0
Create_final_dataset	1	0	0	0	0

Tabla 9: Matriz sin procesar en preprocesado de IDS

Tras el proceso del *collapser* y aplicar las etiquetas:

Grupo1: [main, create_final_dataset, process_piece]

Grupo2: [done, process_piece]

Grupo3: [assign_piece, process_piece]

Grupo4: [process_piece]

	Grupo1	Grupo2	Grupo3	Grupo4
Grupo1	1	0	0	0
Grupo2	1	0	1	0
Grupo3	1	0	0	0
Grupo4	1	0	0	0

Tabla 10: Matriz procesada en preprocesado de IDS

- Splitter:
    - du_0:
        - main
        - create_final_dataset
        - process_piece
        - Cloudbook_thread_counter: Para gestionar los hilos

- du_1:
  - process_piece
  - done
- du_2:
  - process_piece
  - assign_piece
- du_3:
  - process_piece

- Deployer:

```
{
"du_0": ["agent0"],
"du_1": ["agent1"],
"du_2": ["agent2"],
"du_3": ["agent3"]
}
```

4.2.4.2. Análisis de la ejecución

Durante la ejecución de Cloudbook se producen las siguientes comunicaciones entre agentes:

- Función main:
  - 1 llamada a la variable global done
  - 1 llamada a assign_piece
  - 1 llamada a process_piece
  - 1 llamada a create_final_dataset
- Función assign piece:
  - 1 llamada a la variable global done

- El resto de las funciones no interactúan entre ellas.

Como se puede observar, en este problema el paralelismo es de grano grueso, ya que involucra muy pocas comunicaciones, y el tiempo de ejecución estará determinado por la característica que haga que process_piece sea más lento, ya que el resto se ejecutan en paralelo y terminarán antes.

4.2.4.3. Resultados y discusión

A continuación, se muestran los resultados obtenidos tras la ejecución del caso de uso de forma secuencial y en el entorno de Cloudbook:

lineas	secuencial	cloudbook
100000	5,17	4,25
1000000	51,82	25,52
5000000	257,06	139,7
10000000	529,09	300,96
50000000	2737,52	1363,32
157602189	13846,434	6451,53

Tabla 11Resultados preprocesado ids

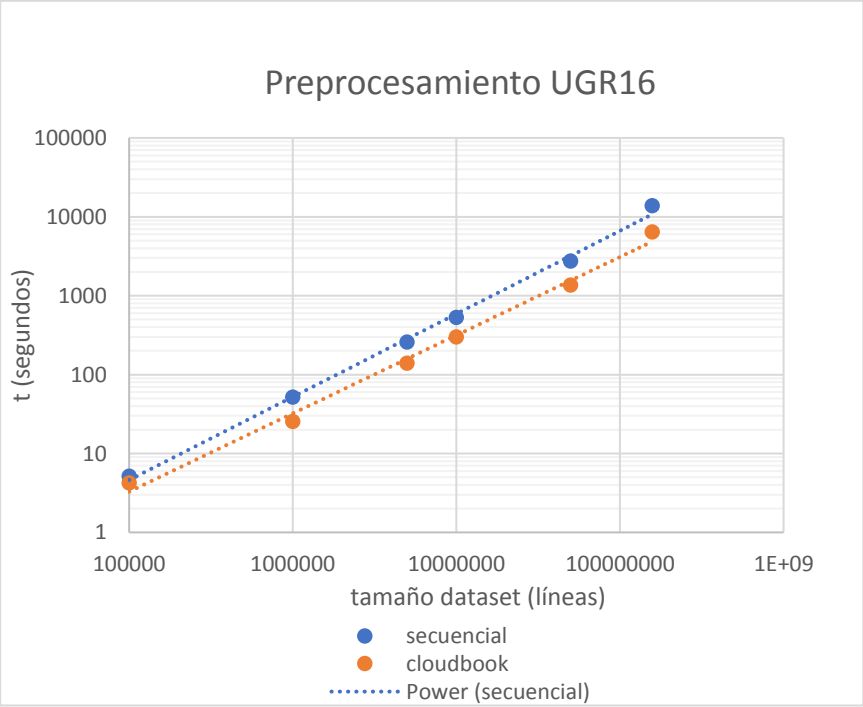


Figura 35: Resultados preprocesado IDS

Como se ve, desde el principio Cloudbook tiene un tiempo igual o menor que la ejecución secuencial, esto se debe a que este problema es de paralelismo de grano grueso, las comunicaciones son mucho menores y sirven para indicar a cada función paralela que parte de los datos procesar, en este caso conseguimos paralelismo desde el principio reduciendo los tiempos hasta la mitad. No se alcanza el paralelismo máximo, porque el procesamiento de los datos no es el mismo en los cuatro agentes, el tiempo total depende del agente que tenga el procesamiento de datos más complejo.

4.3. Conclusiones

Tras la ejecución de estas pruebas de concepto podemos concluir:

- El modelo de ejecución de Cloudbook es válido para producir código paralelo y distribuido de forma automática y permitir de forma transparente la supercomputación.
- Al distribuir el código nos enfrentamos a dos problemas que nos impiden alcanzar un paralelismo ideal (en el que el tiempo de procesado de la parte paralelizable del programa es el tiempo de la misma tarea secuencial dividido por el número de máquinas entre las que ejecutemos):
    - El tiempo de comunicaciones: Que se reduce cuando el conjunto de datos sobre el que trabajamos es suficientemente grande.
    - La carga producida por los hilos en las llamadas paralelas: Se sobrecarga al procesador que además de gestionar la tarea tiene que gestionar los hilos.
- Por lo tanto, el modelo de ejecución de Cloudbook se adapta mejor a los problemas con paralelismo de grano grueso, o absurdamente paralelos, es decir, los que implican pocas comunicaciones y se ejecutan por separado sobre conjuntos de datos disjuntos.
- El modelo de ejecución de Cloudbook se adapta también a los problemas con paralelismo de grano fino, pero solo obtiene beneficio cuando el problema es computacionalmente muy complejo o trabaja con un conjunto de datos de entrada suficientemente grande.
- Es necesario un mecanismo de control de hilos totales para no sobrecargar a los agentes, esto se proporcionará mediante el parámetro MAX_THREADS, que se incluye en el apartado de trabajo futuro.

- Cloudbook aumenta las capacidades computacionales de las que dispone el programa al distribuirse en varias máquinas como se ha demostrado con la pila de llamadas recursivas al distribuir la ejecución en varias máquinas.

5. Conclusiones y trabajo futuro

En este capítulo se presentan las conclusiones obtenidas a lo largo de este trabajo, referentes a la validación del modelo, al rendimiento del mismo, a las características que aporta, y en comparación con otras tecnologías. Tras esto se muestran las líneas futuras que se deberían seguir para mejorar este trabajo.

5.1. Conclusiones

En el estado actual de desarrollo y experimentación de Cloudbook se han obtenido las siguientes conclusiones:

- Con Cloudbook se demuestra que es posible ofrecer computación distribuida troceando de forma automática la parte procedural de un programa
- Cloudbook usando directivas también es válido para producir código paralelo y distribuido de forma automática y permitir de forma transparente la supercomputación.
- El rendimiento producido no alcanza el máximo rendimiento de un programa paralelo pero el balance entre esfuerzo de aprendizaje y capacidad de producir un programa distribuido o paralelo es muy elevado. Es decir, Cloudbook está diseñado para que se pueda usar con programas normales, no hay que aprender un paradigma nuevo de programación ni una implementación nueva de un lenguaje.
- Los límites del rendimiento de Cloudbook vienen dados por las comunicaciones y la gestión de muchos hilos que se pueden encolar esperando su ejecución, estos dos problemas están previstos en el diseño:
    - En el diseño del *splitter* se han diseñado criterios que reducen el número de comunicaciones.
    - El funcionamiento de las variables globales también reduce comunicaciones ya que por defecto se cachean en el agente que las usa, y solo se "refrescan" cuando se indica explícitamente.
    - Para gestionar los hilos y que nunca se encolen demasiadas llamadas a una función paralela, y se sobrecargue la memoria de un agente, se ha diseñado un mecanismo que hace uso de la variable MAX_THREADS que controla el número máximo de hilos que se pueden generar en los agentes. Este mecanismo esta descrito en la sección de trabajo futuro.

- La principal aportación realizada es la división del código de forma automática tomando como unidad mínima la función.
- En cuanto a la supercomputación, Cloudbook sigue el modelo de la computación HTC de forma más versátil y adaptándose a más tipos de problemas, sin forzar al programador a hacer un diseño distribuido del problema. En la siguiente tabla se recogen las aportaciones de Cloudbook frente a este modelo.

Característica	High Performance Computing (HPC)	High Throughput Computing (HTC)	Cloudbook (HTC flexible)
Tipo de problema	Problemas complejos con subtareas no independientes	Problemas fácilmente divisibles en subtareas independientes	Se adapta al tipo de problema
Relación recursos y eficiencia	Maximiza eficiencia	Maximiza los recursos	Maximiza recursos del grid
Trabajo	Mucha computación en poco tiempo	Mucha computación en mucho tiempo	Mucha computación sin tener en cuenta el tiempo
Interconexión	Mucha interconexión	Poca interconexión	Flexible
Descomposición Tareas	Descomposición de grano fino (bucles, instrucciones)	Descomposición de grano grueso (tareas)	Descomposición de grano medio (funciones)

Tabla 12: Cloudbook desde el punto de vista de la supercomputación

- Hay paralelismos entre Cloudbook y tecnologías actuales como CUDA. Las funciones "PARALLEL" de Cloudbook tienen como equivalente las funciones "KERNEL" de CUDA:
  - Ambas son no bloqueantes.
  - Ambas involucran muchos hilos simultáneos.
  - Ambas retornan *void.*
  - Ambas tienen mecanismos de sincronización para esperar al resto de hilos.
- Cloudbook permite el uso de variables globales de forma similar a Spark (sin la gestión de RDDs), esto es, con dos operaciones: transformaciones que operan sobre los datos,

y acciones que devuelven un conjunto de datos nuevo. Pero lo amplía mediante la opción de “refrescar” una variable global tras el proceso de los datos, es decir tras realizar transformaciones. Y permite devolver un conjunto de datos nuevos en otra variable al igual que las acciones en Spark, o actualizar la variable original, accediendo a la función que la gestiona.

5.2. Trabajo futuro

En este apartado se proponen los distintos campos sobre los que trabajar para mejorar el prototipo de Cloudbook, primero se van a mostrar de forma detallada las mejoras más importantes o urgentes que se tienen que hacer, finalmente en otro subapartado se nombran el resto de las mejoras que hacer.

5.2.1. Ampliar el prototipo al lenguaje completo de Python

En el prototipo presentado en este trabajo, se ha cubierto parcialmente el lenguaje Python. Para asegurar que se puede usar con cualquier programa es necesario que el mecanismo de análisis del código fuente pueda tratar con todas las características de Python, siendo las principales:

- Programación orientada a objetos: Cloudbook actualmente no *trans-compila* los programas que usan abstracción en clases y objetos. Cloudbook analiza la parte procedural del programa, y si este usa clases las debe distribuir en cada unidad desplegable, de forma que cualquier función pueda tratar con un objeto. La traducción en las unidades desplegables debe tener en cuenta las características de la programación orientada a objetos como la herencia y el polimorfismo.
- Decoradores de función, método y clase: Cloudbook actualmente no los tiene en cuenta, pero debería integrar la función decoradora en cada unidad desplegable que contenga un elemento que use un decorador.
- Otras capacidades de Python que no traduce actualmente como estructuras complejas de datos, descriptores, metaclases, y metafunciones.

5.2.2. Cloudbook en modo servicio

En este trabajo se ha presentado Cloudbook en modo local y se ha presentado su funcionamiento como servicio, en este apartado se presenta el diseño del modo servicio y se definen los cambios necesarios para llevarlo a cabo.

Los distintos servicios de Cloudbook estarán en Amazon Web Services usando la tecnología Elastic Beanstalk, la IaaS de AWS. En modo servicio con posibilidad de tener círculos de miles de máquinas NFS ya no es la solución adecuada, se recomienda usar un sistema de ficheros basado en P2P.

Serán necesarias nuevas bases de datos para guardar la información sobre círculos que pueden estar distribuidos por todo el mundo. Las tablas nuevas serán:

- CIRCLES_TABLE: Con los campos circle_id (primary key),status, descripción, número de agentes
- AGENTS_TABLE: Con los campos agent_id (primary key), circle_id, attach timestamp.

5.2.2.1. Circle manager service

Este componente permite la creación y edición de círculos de máquinas en los que se ejecutaran los programas distribuidos. La información del círculo es necesaria para ejecutar el *maker*. Sus funciones son:

- Obtener información de un círculo, o de todos los círculos que hay.
- Apuntar agente a un círculo (e indicarlo en la tabla AGENTS_TABLE)
- Modificar círculo
- Obtener agentes de un círculo

5.2.2.2. Cambios en los agentes

Para que los agentes puedan comunicarse correctamente es necesario que todos los agentes que pertenecen al mismo círculo cuando se den de alta con el gestor de círculos publiquen su IP pública y su identificador, esta operación se repetirá cada cierto tiempo y funcionará como un *heartbeat* que indicará qué agentes están disponibles.

5.2.2.3. Ip publisher service

El módulo de publicación de direcciones IP. Este módulo recibe información de los agentes, su id, y su IP pública y las guarda. Tareas:

- Mantiene IDs e IPs de cada agente actualizada.
- Cuando es requerido por uno de los agentes o el *deployer*, indica que agentes están actualizados y ejecutándose.

5.2.2.4. Deployer service

Este servicio se comunica con el servicio de gestión de círculos para saber que agentes forman parte del círculo y poder construir el fichero *Cloudbook.json.* Se comunica con el *Ip publisher service* para iniciar la ejecución en la unidad desplegable 0, que contiene el main.

5.2.2.5. Maker Service

El *maker* como servicio funciona como en local, pero interactúa con la taba CIRCLES_TABLE actualizando el campo status a “make on going”.

5.2.3. Redespliegue dinámico y tolerancia a fallos

En su estado actual, Cloudbook tiene dos carencias en cuanto al despliegue de la plataforma. El despliegue se basa en un análisis estático que puede estar equivocado y perjudicar el rendimiento de los programas, y no es tolerante a fallos, esto es, si una máquina se cae, la ejecución falla. En el futuro es necesario que estas dos carencias se resuelvan y en este apartado se muestra el diseño de una solución y cómo se aplica a la plataforma en el caso de que se ejecute Cloudbook en modo local, el presentado en este trabajo, y en modo servicio, diseñado también en este capítulo.

5.2.3.1. Tolerancia a fallos. Comportamiento de los agentes y deployer

Los agentes deben publicar periódicamente su dirección IP, esto dependiendo del modo en el que se use Cloudbook será:

- Modo servicio: En el módulo *ip_publisher_service*.
- Modo local: Añadiendo una entrada en el fichero *local_ip_info.json*

El *deployer* deberá comprobar el servicio o el fichero para validar que los agentes están en ejecución. En caso de que un agente no esté se deberá redesplegar, es decir, generar un nuevo fichero *Cloudbook.json*.

En este punto hay que tener en cuenta el periodo de comprobación del *deployer* ha de ser mayor que el periodo de publicación de IP.

5.2.3.2. Mejora de eficiencia. Comportamiento de los agentes

Para medir la eficiencia es necesario iniciar con el agente un proceso en segundo plano, que mida las estadísticas que se describen a continuación:

- Sobre las invocaciones externas: Número de invocaciones entre funciones, este dato nos ayuda a rellenar la matriz de invocaciones con datos reales. Se guarda la información como una tupla

  <(du_x, f_invoker, f_invoked, contador, tiempo respuesta medio)>

- Sobre las funciones propias: Las funciones etiquetadas como PARALLEL o NONBLOCKING generan una tupla incluyendo el tiempo de respuesta media, ya que desde el invocador este tiempo es cero.

- Uso de recursos computacionales por el agente: utilizando la librería de Pyton *msutil* se medirá:

  - Número medio de threads lanzados
  - Porcentaje medio de uso de CPU
  - Media de la memoria Ram disponible

Estas estadísticas se guardarán en un fichero que se aumentara cada cierto periodo de tiempo con las últimas estadísticas recolectadas.

5.2.3.3. Mejora de eficiencia. Comportamiento del maker y deployer

Para realizar un redespliegue basado el rendimiento actual es necesario que la matriz de invocaciones sea accesible en el sistema de ficheros distribuido, esta matriz, solo con las funciones, sin haber colapsado nada se debe escribir en el fichero *matrix.json*.

Debe haber un proceso en el módulo *deployer* que analice las estadísticas de la ejecución y genere una nueva versión de la matriz con los datos reales de las invocaciones.

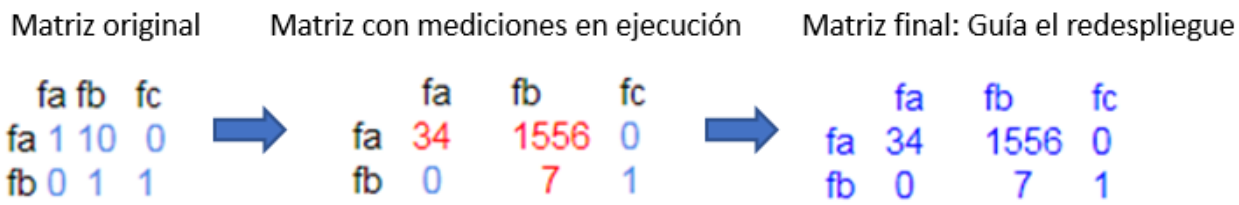


Figura 43:Evolución de la matriz de invocaciones

Con la nueva matriz hay que invocar al *splitter* para que colapse las funciones en función de los datos reales y genera las nuevas unidades desplegables. La primera vez que se obtiene la nueva matriz se realiza la generación de unidades desplegables, pero el resto de las veces que se calcule la matriz, solo se generarán nuevas unidades desplegables cuando haya un cambio como mínimo de un orden de magnitud en un valor.

Cuando se generen las nuevas unidades desplegables hay que crear de nuevo el fichero *Cloudbook.json* para que se haga efectivo el redespliegue.

5.2.3.4. Uso de métricas para validar la distribución

Se va a definir la métrica PR, *performance ratio* que está basada en:

- Trabajo: Suma de las invocaciones durante un periodo por su coste

W = (n1*cost_f1 + … + nN*cost_fN)

Donde:

n = número de invocaciones

fx = la función invocada

- Número de máquinas = M
- PR = W/M = ratio of performance

PR mínimo =1, máximo = infinito

PR es una medida de rendimiento, que crecerá tras varios redespliegues hasta llegar a su máximo. La medida PR no se usa para tomar decisiones. Para tomar decisiones, se usa la matriz cuya precisión es exacta.

5.2.3.5. Detección de redespliegue

Si un agente realiza una petición y no recibe respuesta, esto puede ser porque haya habido, o haya en ese momento un redespliegue, por lo que debe esperar un periodo de tiempo y comprobar el fichero *Cloudbook.json*. Este periodo de tiempo debe ser configurable en la configuración global del círculo, *config.json* ya que el tiempo de despliegue depende del tamaño del círculo.

5.2.3.6. Sumario de cambios propuestos

- Agente
    - Los agentes publican periódicamente su IP en el "*ip publisher service"*.
    - El agente publica periódicamente las estadísticas que ha recopilado.
    - Si otro agente no responde a una invocación, el agente comprueba si ha habido un redespliegue.
    - Los agentes comprueban periódicamente *Cloudbook.json* por si ha habido un redespliegue.
- Deployer
    - El *deployer* periódicamente comprueba las estadísticas y decide si invocar al *splitter*.
    - Tras la terminación del *splitter* el depoyer debe producir un nuevo *Cloudbook.json.*
    - En modo local el *splitter* y *deployer* deben estar en la misma máquina para ser invocado de forma sencilla.
- Maker
    - Matrix filler guarda la matriz en el sistema de ficheros distribuido.
    - El *splitter* puede ser invocado desde el *deployer*.

5.2.4. Mejoras de la plataforma

- Implementar el mecanismo de gestión de *threads* con la variable MAX_THREADS, este mecanismo ya ha sido diseñado y consiste en meter las llamadas *parallel* en un bucle controlado por el número de *threads* lanzados, cuando hay vivos más de

MAX_THREADS el programa espera en el bucle a que alguno termine antes de lanzar otro.

    - Esta característica permite que no se sobrecarguen los agentes por encolar demasiadas llamadas a funciones *parallel* y consuman mucha memoria.
    - Se puede saber cuántos hilos se han lanzados gracias a la función *Cloudbook_max_threads.*

- Para no sobrecargar agentes con trabajo cuando gestionan funciones paralelas, se podría evitar que la unidad desplegable que realiza las invocaciones a funciones paralelas, no las ejecute, ya que tiene más "trabajo" enviando peticiones y ejecutando, que el resto que solo ejecutan.
- Explotar el paralelismo implícito de los programas: Las funciones con la etiqueta __*Cloudbook:Local*__ se pueden detectar en la matriz e implementar de forma automática sin que el programador ponga la etiqueta. Otras formas de paralelismo se pueden detectar.
- Implementar una etiqueta nueva que permita agregar llamadas paralelas, para poder realizar llamadas a funciones paralelas con más datos y así, aumentar la granularidad del paralelismo y mejorar la eficiencia, en lugar de recomendar al programador agregar los datos el mismo, si quiere explotar mejor el paralelismo.
- Mejorar los protocolos de comunicación entre agentes usando protocolos que ofrezcan mejor rendimiento.

5.2.5. Nuevos casos de uso

Para explotar la adaptabilidad de Cloudbook a cualquier tipo de programa se va a probar también con dos nuevos casos de uso con aplicaciones muy distintas:

- Aplicaciones *VideoWall*: Estas son aplicaciones diseñadas para ser ejecutadas en un *grid* de pantallas, como puede ser el mapa de un videojuego como se ve en la Figura 44

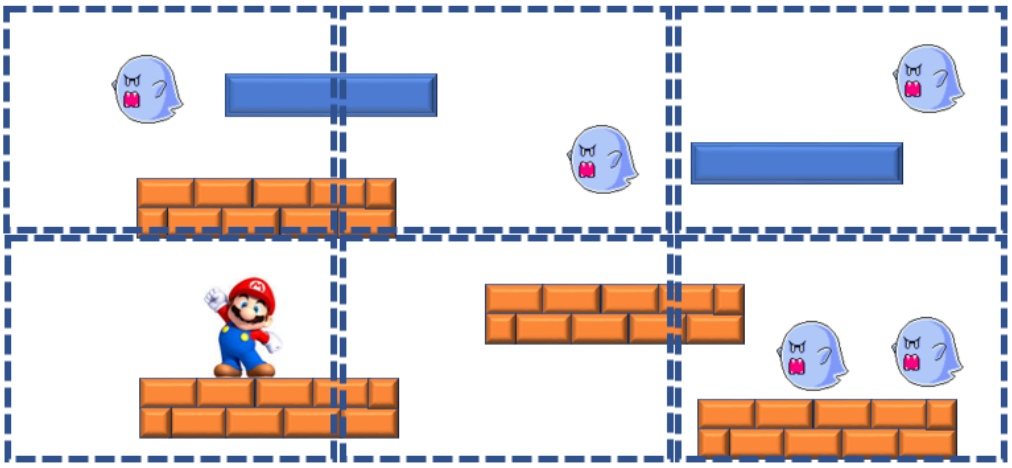


Figura 44: Videojuego distribuido en 6 pantallas

El objetivo de la aplicación es conseguir distribuir el *rendering* y la lógica del videojuego en N máquinas. Una aproximación de la estrategia es:

  - *Rendering*: El juego puede representar gráficos en una pantalla virtual enorme, y asignar partes de esa pantalla virtual a las pantallas físicas disponibles.
  - Lógica de personajes: Una posible implementación es distribuir las funciones de lógica de los personajes en todos los agentes, con la restricción de que solo ejecutan los que aparezcan en las coordenadas que tiene que reproducir en su pantalla.
  - Funciones generales: Las funciones que no tienen que ver con los personajes, como la colisión de sprites, control de puntuaciones, podrían distribuirse solo en un agente, y ser invocadas cuando sea necesario.

- Videojuegos multijugador sin servidor: Otro posible caso de uso sería el de implementar las funciones de juego en red de un juego, sin la necesidad de usar los paradigmas actuales, que son multijugador con servidor, y multijugador p2p, la aproximación en Cloudbook (véase Figura 45) se acerca más al multijugador p2p, ya que no tiene servidores.

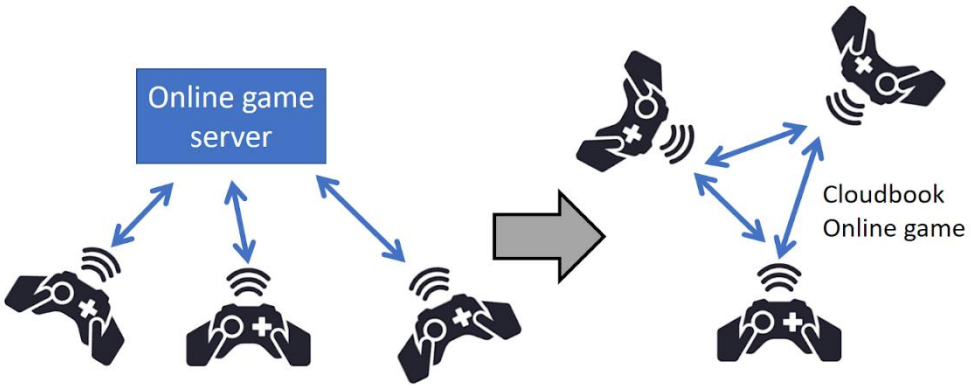


Figura 45: Propuesta multijugador Cloudbook

Una posible implementación del mismo sería:

- Lógica y *rendering*: La lógica del juego estaría en cada agente, y se representaría solo el punto de vista del jugador que se despliega en cada agente.
- Sincronización del status del juego: Las tablas de puntuaciones se usarán como una variable global gestionada en tiempo de ejecución.
- El juego propuesto es un juego simple multiusuario en 2D, en el que los jugadores se pueden mover de forma libre por un mapa y tienen que recolectar recursos, que serán una variable global gestionada por Cloudbook.

6. Introduction, objectives and document structure

This chapter introduces the project presented in this work, explaining the objectives to be achieved, also explains the methodology of work followed, and finally explains the structure of this document.

6.1. Introduction

As processors become faster and more effective, physical restrictions have appeared that prevent processors from continuing to scale in frequency (1). At the same time, the reduction of energy consumption has gained importance in recent years (2) due to this parallel programming has become the paradigm to follow in computing, both at the level of multiprocessors in home computers (2) and for supercomputing. As has happened with distributed computing, necessary to manage telecommunications networks, networked applications, real-time access protocols and parallel computing itself (3).

These computational needs have given rise to many technologies that exploit parallelism to improve efficiency, maximize resources and reduce costs and consumption: OpenMP, CUDA on graphics cards; the distribution: BOINC, Charm++; or both: Apache Spark and Apache Flink in the field of Big Data analysis.

This is why computing paradigms that mix parallel and distributed computing are increasingly necessary, in this context arises grid computing, clustering, and cloud computing.

Grid computing is a type of distributed computing in which a virtual supercomputer is composed by the weakly coupled union of many computers working together to perform a complex task, this work can be done simultaneously or sequentially (4).

These technologies face two main problems, the design of code so that it can be distributed or parallelized, and the distribution and execution of previously prepared code in different architectures, or machines in different networks.

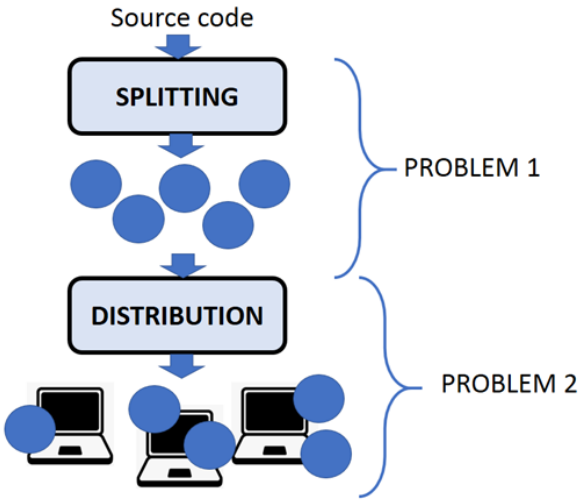


Figura 46: Problems Faced

The first problem can be addressed in two ways, either by creating code specially prepared for future distribution and parallel execution, or by automatically transforming the code. In the first case a lot of knowledge is required of the programmer since parallel algorithms are more difficult to write (5), and the second case requires very fine grain control over the source code and the compiler.

In this work another possibility is offered to take advantage of the distribution and parallelism without demanding to the programmer a change of paradigm in the programming language or to carry out a design of the distributed program.

For this purpose, a platform will be designed and developed that, using as a minimum processing unit the program's functions instead of loops or data, carries out a trans-compilation process that translates the source code of a program into code that allows the platform to distribute and execute it transparently, following the grid computing model, in order to offer a supercomputer made up of all distributed computers (see Figure 2).

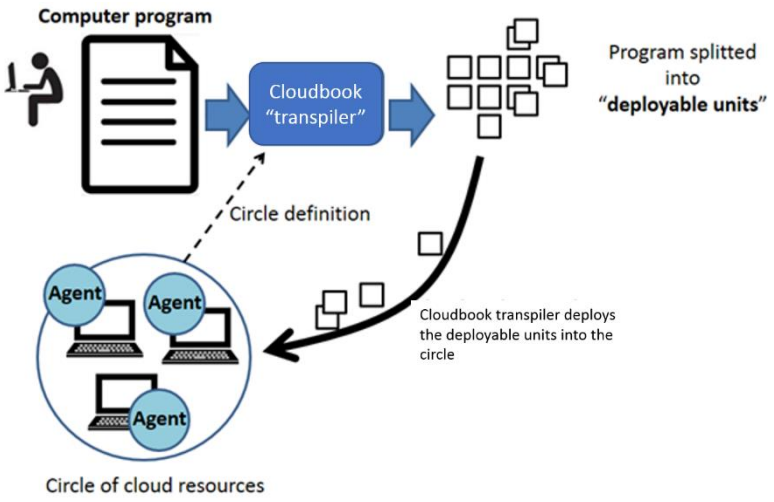


Figura 47: Cloudbook Diagram

6.2. Objectives

In this work we are going to design, develop and test a platform that allows a programmer to make automatically distributed programs abstracting from the traditional paradigms of distributed programming. It will also provide functionality that allows parallelism to be exploited during the execution of the program.

This will free the programmer from a complex design oriented to distribution, without renouncing the advantages of distributed or parallel computing with a very low learning cost.

In order to achieve this goal, two problems have to be solved, these are the automatic analysis of the code and the production from it of the code destined for distribution, and the distribution itself in the network of machines in which it will be executed.

6.2.1. Objective 1: Analysis and automatic generation of source code

The first objective of this work responds to the need for automatic code generation [7] oriented to parallelization and distribution, to better harness the power of processors, and the automatic distribution of it.

The platform will allow that, given a program, it can be automatically divided in deployable pieces (deployable units) that in turn are distributed for its execution in a given set of processors, and if the programmer wants, to use labels that allow to take advantage of the parallelization.

To do so, it will restructure the code according to a defined criterion and will display it on different computers that interact following the order foreseen by the programmer. One of the advantages of this system is that the code of the original program does not need to include special sentences for its distribution nor is it necessary to use a specific parallel programming language for its development. In addition, the deployable units will be able to communicate with each other without the need for a central orchestrator.

The analysis of the code will generate a graph of dependencies between the functions of the original program and will distribute them in different deployable units depending on the distributed environment where it will be executed and on different criteria such as the size of the deployable units or the communication capacity in the network.

Code will be generated automatically to allow communication between different deployable units, manage global variables, and allow parallel execution of certain functions designated by directives.

6.2.2. Objective 2: Distribution of source code

The second objective of this work consists in the deployment of the code in the different computers defining sets of these, called circles, which will have access to the source code and input data in a distributed file system and taking into account the needs of the network, such as Internet access to private networks with address translation (NAT).

The circles will consist of different computers, each of which will run an agent, who will be jointly responsible for the execution of the distributed program.

The agents are processes that must be started in a port of the computer and wait to know which deployable units correspond to them to execute. In addition, they must be able to invoke functions in other agents.

The agents must know the location of the distributed file system in order to be able to access the source code they must execute, know the addresses of the rest of the agents belonging to the circle in order to be able to call their functions, and mainly know which deployable units have been assigned to them.

The distributed file system will not be taken into account in the development, leaving it to the user's choice. The only action you must take is to indicate where it is so that the platform works with the source code and data found there.

6.2.3. Objective 3: Analyse the platform and its use

A functional prototype will be developed to validate proofs of concept. This will make it possible to analyse the behaviour of the platform, validate the hypotheses put forward and effectively measure the advantages of the platform.

Another objective within this is to ensure that the platform is not a challenge for users and can be used without specific knowledge about software distribution.

To this end, the tests will be carried out with code developed for a non-distributed execution, as the distribution will be carried out automatically by the platform.

The tests will be tested with the platform, without it, with the aim of providing an analysis of performance and ease of use that mark the strengths and weaknesses of the platform and the cases in which its use can be more beneficial.

6.3. Work methodology

The methodology used in this project is based on the definition of an objective for the project and a study of the state of the art to identify related works, what problems occur in the area studied, and what can be contributed to it, and thus provide a more concrete definition of the objective of the project.

After this, a series of hypotheses are realized that allow the iterative development of the platform that constitutes the project.

The iterative design of the platform is carried out in different stages:

1. Generic architecture proposal, which is the one observed in section 2 of chapter 3 of this report.
2. The architecture is validated by developing a simple prototype of each component.
3. The different hypotheses are tested on each component in order to show their validity and whether it is necessary to apply new functionalities.
4. The prototypes of the components can interact to validate the use throughout the platform, thus validating the previous steps and producing proposals for improvement or new hypotheses for the following iterations.

This iterative design is shown throughout the description of the components in section 3 of chapter 3. And it concludes with the prototype presented in this project. Future improvements to the architecture are described in chapter 5.

6.4. Document structure

This work has been structured in the following chapters:

- Chapter 1: Introduction and objectives

This chapter presents the platform on which the work is based, introducing the problem on which it is working and its motivation. It also states the objectives of the work and explains the methodology followed to carry it out.

- Chapter 2: State of the art

In this chapter we study the current technologies related to this work and discuss our contribution to the state of the art.

- Chapter 3: Cloudbook Platform

This chapter consists of three parts: The first, a description of the technologies used for the development of the Cloudbook platform. The second, a description of the high-level project explaining how the platform works. And the third, a detailed description of the elements that make up the platform and how they work.

- Chapter 4: Experimentation and results

This chapter presents different evidence on the platform developed, which addresses different cases of parallelization and distribution. The results obtained are also discussed according to their suitability to the proposed objectives.

- Chapter 5: Conclusions and future work

This chapter shows the conclusions obtained in the development and testing of the platform and details the future lines of work on which to continue implementing the platform.

- Annexes: Programming manual

Also included in this work is a guide to installing and using the platform, along with programming recommendations.

7. Conclusions and future work

This chapter presents the conclusions obtained throughout this work, referring to the validation of the model, its performance, the characteristics it provides, and in comparison, with other technologies. After this, the future lines that should be followed to improve this work are shown.

- With Cloudbook it is demonstrated that it is possible to offer distributed computing by automatically cutting up the procedural part of a program.
- Cloudbook using directives is also valid to produce parallel and distributed code automatically and allow transparent supercomputing.
- The performance produced does not reach the maximum performance of a parallel program but the balance between learning effort and the ability to produce a distributed or parallel program is very high. In other words, Cloudbook is designed to be used with ordinary programs, there is no need to learn a new programming paradigm or a new language implementation.
- The limits of Cloudbook performance are given by the communications and management of many threads that can be glued waiting to be executed, these two problems are foreseen in the design:
- In the design of the splitter have been designed criteria that reduce the number of communications.
- The operation of the global variables also reduces communications since by default they are cached in the agent that uses them, and they are only "refreshed" when explicitly indicated.
- To manage the threads and that never too many calls to a parallel function are queued, and the memory of an agent is overloaded, a mechanism has been designed that makes use of the MAX_THREADS variable that controls the maximum number of threads that can be generated in the agents. This mechanism is described in the future work section.
- The main contribution made is the division of the code automatically taking as minimum unit the function.
- As for supercomputing, Cloudbook follows the model of HTC computing in a more versatile way and adapts to more types of problems, without forcing the

programmer to make a distributed design of the problem. The following table shows the contributions of Cloudbook to this model.

Feature	High Performance Computing (HPC)	High Throughput Computing (HTC)	Cloudbook (HTC flexible)
Type of problem	Complex problem with non-independet subtasks	Easilly divisible problems into independent subtasks	Adapts to the type of problem
Resources and efficiency ratio	Maximizes efficiency	Maximizes resources	Maximizes grid resources
Workload	A loto f computing in a short time	A lot of computing in a long time	A lot of computing regardless of time
Interconnection	A lot of interconnection	Little interconnection	Flexible
Tasks decomposition	Fine grained decomposition (loops, instructions)	Coarse grained decomposition (tasks)	Medium grain decomposition (functions)

Tabla 13: Cloudbook from the point of view of supercomputing

- There are parallels between the Cloudbook and current technologies such as CUDA. The "PARALLEL" functions of the Cloudbook are equivalent to the "KERNEL" functions of CUDA:
  - Both are non-blocking
  - Both involve many simultaneous threads
  - Both return void
  - Both have synchronization mechanisms to wait for the rest of the threads
- Cloudbook allows the use of global variables in a similar way to Spark (without RDDs management), that is, with two operations: transformations that operate on the data, and actions that return a new set of data. Cloudbook extends it by means of the option of "refreshing" a global variable after the data processing, that is to say after carrying out transformations. And it allows to return a set of new data in

another variable like the actions in Spark, or to update the original variable, accessing the function that manages it.

7.1. Future work

In this section we propose the different fields on which to work to improve the prototype of Cloudbook, first they will show in detail the most important or urgent improvements that have to be made, finally in another subsection the rest of the improvements that must be done.

- Extend the prototype to the full Python language. It is necessary to extend the platform to accept the following features:
    - Object-oriented programming.
    - Decorators of function, method and class.
    - Other capabilities such as complex data structures, descriptors, *metaclasses* and *metafunctions*.
- Cloudbook in service mode: Develop Cloudbook as an internet service that can be accessed to participate in public circles.
- Allow dynamic redeployment and fault tolerance: Allow the Cloudbook to react to a fault in a machine in your circle, and also to reorganize deployable units to improve performance.
- Other necessary improvements:
    - Implement the MAX_THREADS mechanism to control the threads launched.
    - Allow that the functions that call parallel functions do not assign to themselves the execution of the same ones in order not to overload the agents.
    - Exploit the implicit parallelism of the program, automatically detecting the functions that are labeled as__*Cloudbook:Local*__
    - Implement a new label that allows the addition of data in parallel calls.
    - Improve communication protocols between agents using protocols that offer better performance.

- New use cases:
    - Videowall applications: In which the management of each screen that composes the vodeowall is distributed.
    - Multiplayer games without server: In which the network management is distributed among the different players, who are the agents.

Apéndice A - Manual de programación de Cloudbook

Introduction

A brief definition of Cloudbook:

- Cloudbook is a kind of "transcompiler" which can transform a standalone program into a distributable-parallel program.
- Cloudbook input may include some Cloudbook pragmas for optimal Cloudbook results. These pragmas and some examples are described in this document.
- Cloudbook supports python programs but its concepts may be port to any languages.

Cloudbook installation and config (Local Mode)

Cloudbook will execute your python program using a set of machines. At least you should have 2 machines to check the results.

Setup the environment

These are the steps to do before Cloudbook installation procedures:

- install the distributed filesystem in the target machines.
- create a folder Cloudbook with subfolders "source" and "output"
- copy your program (source code) into the source folder

Following directories should be created:

- windows: $HOMEDRIVE/$HOMEPATH/Cloudbook/
- linux: /etc/Cloudbook/

Install the environment

Prerequisites:

- You need python2 and python3
- Install ply (at least in the machine in charge of run Cloudbook_maker)
- Install flask, pynat, urllib

- Install radon

In each machine, you must install the Cloudbook agent, downloadable from

- https://github.com/Cloudbook-project/Cloudbook_agent
- Attach every agent to the default circle
- Configure the circle to use the "source" folder of the shared Cloudbook directory

In the machine in charge of the maker, install maker and deployer

- Maker: https://github.com/Cloudbook-project/Cloudbook_maker
    - Configure maker to use the shared directory
        - By default:
            - windows: $HOMEDRIVE/$HOMEPATH/Cloudbook/config.json
            - linux: /etc/Cloudbook/config.json
    - Configure maker variable CLOUDBOOK_MAXTHREADS to the desired value. By default is 10. This variable represents the maximum number of alive invoked functions simultaneously by an agent. This affects performance. You can leave the default value and adjust later at program execution tunning phase. This variable is located at Cloudbook/config/config_maker.json
- Deployer: https://github.com/Cloudbook-project/Cloudbook_deployer
    - Configure deployer to use the shared directory
        - By default:
            - windows: $HOMEDRIVE/$HOMEPATH/Cloudbook/config.json
            - linux: /etc/Cloudbook/config.json

You are ready

1. Execute maker, just type:

   >python cloudbook_maker.py

2. Check your deployable units are created, and no errors are in the console
3. Start your agents

   >python3 gui.py
4. Generate Cloudbook.json

   >python cloudbook_deployer.py
5. Execute deployer to invoke main() on the agent in charge of DU0

   >python3 cloudbook_run.py

Beginner

Cloudbook maker is a command to process your program and generate an output composed of several pieces of code called deployable units.

In detail:

- Cloudbook analyzes the procedural structure (not object-oriented "classes") of your program and split this structure into a set of interoperable pieces of code, called "deployable units". Each deployable unit is a python file named "DU_<number>". The number of DUs may not be "complete". It is possible that the output of Cloudbook contains DU0, DU3, DU4 but not DU1, DU2. The only important point is that any DU has a unique name.

The number of desired DUs is configurable through file: Cloudbook/config/config_maker.json

- Cloudbook is able to parse functions with any number of parameters
- Cloudbook support different pragmas (or labels) to produce an efficient distributed parallel code

Cloudbook pragmas

Cloudbook supports the following language extensions:

- pragmas at function definition :
  - **#__CLOUDBOOK:NONBLOCKING__**: functions with this label can not return anything. When Cloudbook detect a nonblocking

function, its code is modified to launch a thread at the invoked agent and returns immediately. These functions can not return any value

- **#__CLOUDBOOK:PARALLEL__**: these functions are deployed in all DUs. These functions are **nonblocking** by construction and therefore are not allowed to return anything. The difference between NONBLOCKING and PARALLEL consist of the number of DUs in which the function is deployed. Nonblocking functions are deployed in only one DU, whereas parallel functions are deployed in all DUs
- **#__CLOUDBOOK:PARALLELNONSYNC__**: these functions are like PARALLEL ones but not synchronizable. It means that the SYNC pragma doest allow waiting for the termination of all launched functions.
- **#__CLOUDBOOK:RECURSIVE__** : these functions are deployed in all DUs . The behaviour is defined to maximize the level of recursivity. Each recursive invocations from any DU invokes other DU, which means that in a circle with 10 machines you have 10 times more recursive level than in one machine.
- #**__CLOUDBOOK:LOCAL__** : these functions are deployed in all DUs, in order to be available for local invocations, avoiding communications. This pragma is intended to be considered at "tunning" phase of the program.

- pragmas at function invocation
  - **#__CLOUDBOOK:NONBLOCKING__**: If the function is not defined as NONBLOCKING but the programmer does not want to wait for its execution, can invoke the function using this label. in this case, a thread is launched at invoker agent, whereas when the label is used at function definition, the thread is created at the invoked agent
  - **#__CLOUDBOOK:SYNC__:** wait for all launched non-blocking operations to be ended. This pragma does not allow synchronize PARALLELNONSYNC functions

Cloudbook support global variables, but special treatment is needed

- **global:** this python keyword indicates to Cloudbook that must refresh the value of global var. Otherwise, a local cache copy of the var is used. The use of a local copy benefits the performance, reducing communications.

IMPORTANT: note that "global" is not a Cloudbook pragma, but a python keyword

- **#__CLOUDBOOK:SAFE__ :** this pragma before a global variable creation, indicates Cloudbook to treat this variable in "safe mode". Safe mode avoids any possibility to have different values of the same variable at different agents/threads. The performance is damaged because this mechanism is based on a read/write lock and any access to the variable is translated into exclusive access. **When read the lock is set, and after any modification, the lock is released**

Hello world

Cloudbook needs to find only one main function in your program files. However, python allows having an implicit main function in every file. Do not use this python feature, and create a unique main function, this can be main(). There can only be one main function in the file that you want to direct the execution or use a unique main function, labelled with "#CLOUDBOOK: main()"

```
def main():
    print ("hello world")
#CLOUDBOOK: main()
main()
```

Global variables

Use always the “global” qualifier (python keyword) inside the functions that use the var. IMPORTANT: note that “global” is not a Cloudbook pragma, but a python keyword

```
myvar =0
def main():
    global myvar
   print (myvar)
main()
```

- Cloudbook modify all functions code to work with an updated value of the global variable just at the beginning of the function. Therefore you can be sure that your function **works with a fresh value** of the global variable just when the function execution starts
- During function execution, the variable accessed for reading is not the global var, **but a cached copy of it (it is not a fresh value)**
- Cloudbook updates the global var storage when the programmer uses an instruction to change its value, such as “append” (in case of lists) or “=” (in case of integers, etc.). Each change involves communications with the remote agent in charge of global variable storage and therefore is recommended **to minimize this type of operations.**

Safe global variables

Safe global vars are built using #__CLOUDBOOK:SAFE_ pragma. This pragma before a global variable creation indicates Cloudbook to treat this variable in “safe mode”.

Safe mode avoids any possibility to have different values of the same variable at different threads. The performance is damaged because this mechanism is based on a read+write locks and any access to the variable is translated into exclusive access. Hence the locks may produce thread blocking, **it is forbidden** to lock more than one safe global variable at the same time.

When read the lock is set, and after any modification, the lock is released

at main program:

#__CLOUDBOOK:SAFE__

global A # creates global “safe” variable A

#__CLOUDBOOK:SAFE__

global B # creates global “safe”variable B

at any agent :

global A # lock and reflesh A value

A = A+1 # modifies A, and then publish A value and unlock A value

IMPORTANT: safe global variables may produce interlocking effects. Therefore the following program is forbidden and raise an error at Cloudbook compilation. **You can only lock one variable inside any function**

agent 1	Agent 2
global A #locks A	global B #locks B
global B # waits for B	global A # waits for A
...	…

IMPORTANT: keep in mind that safe global variables penalize performance and must be used only when any other option (a normal global variable) is not suitable for program purposes.

Tunning maximum performance

Performance using parallel

In order to take benefit from all machines belonging to a circle, there are a lot of recipes. The main one is:

1. Declare a functionA() as "**PARALLEL**": the function will be deployed in all DUs of the circle. Functions labelled as "parallel" are non-blocking (Cloudbook modifies its code to launch a thread and return immediately). Remember that "parallel" functions can not return any value.
2. Invoke massively functionA() .

Doing both, the functionA() will be executed in parallel by all DUs, because each invocation to a parallel function is directed to a different machine (round robin), and the nonblocking feature of parallel functions, allows to create new requests while the previous invocations are being executed
Note that the number of threads created in a loop invoking the parallel function is not greater than **CLOUDBOOK_MAX_THREADS**

```
# CLOUDBOOK:PARALLEL
def functionA():
    for x in 100:
            <do something>
#inside main()
for i in 10000:
    functionA()
```

The maker variable CLOUDBOOK_MAXTHREADS allows to launch up to CLOUDBOOK_MAXTHREADS functions in parallel and waits to launch the next one until any of the previously launched functions ends

This variable is important if you want to optimize your resources. If you have 4 machines of 4 cores, launch more than16 functions simultaneously does not produce more performance, but damages it.

Performance using parallel and aggregation of data

It is possible to launch 10000 functions in parallel, each one launch a thread in a different agent. However, is more efficient to reduce the number of total existing threads, saving memory resources. To achieve this efficient invocation we must consider to have functions with range parameters.

For example, in this case the total number of simultaneous threads is 10000

```
# CLOUDBOOK:PARALLEL
def functionA():
    for x in 100:
            <do something>
#inside main()
for i in 10000:
    functionA()
```

The example could be rewritten in a better efficient way, using range parameters at functionA definition and invoking 100 times instead 10000 times. In this case, the total number of threads is 100

```
# CLOUDBOOK:PARALLEL
def functionA(ini, fin):
     for j in range(ini,fin-1):
       for x in 100:
          <do something>
#inside main()
for i in range (0,10000,100):
    functionA(i,i+100)
```

Referencias

1. Adve, Sarita y al, et. *Parallel @ Illinoiis.* Illinois : University of Illinois, 2008.

2. Asanovic, Krste y al, et. *The Landscape of Parallel Computing Research: A.* s.l. : Berkeley University, 2006.

3. Andrews, Gregory. *Foundations of Multithreaded, Parallel, and Distributed Programming.* s.l. : Addison-Wesley, 2000.

4. Magoulès, Fréderic. *Fundamentals of grid computing.* s.l. : CRC-Press, 2010. Capítulo 1 Seccion 4.

5. Hennessy, John L y Patterson, David A. *Organización y diseño de computadores: la interfaz hardware/software.* 1994.

6. Tanenbaum, Andrew S y Steen, Maarten van. *Distributed systems: Principles and paradigms.* s.l. : Pearson Prentice Hall, 2002.

7. Godfrey, Bill. A primer on distributed computing. [En línea] 2002.

8. Herlihy, Maurice P. y Shavit, Nir N. *The Art of Multiprocessor Programming.* s.l. : Morgan Kauffmann, 2008.

9. *Real Time And Distributed Computing Systems.* AL, Kamble, Shindle, TK y N, Kothiwale, SS, Khot. 2012, IOSR Journal of Computer Engineering, págs. 53-56.

10. Lynch, Nancy A. *Distributed Algorithms.* s.l. : Morgan Kauffman, 1996.

11. Ghosh, Sukumar. *Distributed Systems – An Algorithmic Approach.* s.l. : Chapman & Hall CRC, 2007.

12. Peleg, David. *Distributed Computing: A Locality-Sensitive Approach.* s.l. : SIAM, 2000.

13. *news distributed computing column 32: the year in review.* Keidar, Idit. 4, 2008, ACM SIGACT News, Vol. 39, pág. 53.54.

14. Miym. Distributed-Parallel Computing. [En línea] https://en.wikipedia.org/wiki/Distributed_computing#/media/File:Distributed-parallel.svg.

15. Almasi, GS y Gottlieb, A. *Highly Parallel Computing.* s.l. : Benjamin-Cummings, 1989.

16. Barney, Blaise. Introduction to parallel computing. [En línea] https://computing.llnl.gov/tutorials/parallel_comp/.

17. Rauber, Thomas y Runger, Gudula. *Parallel Programming: for Multicore and Cluster Systems.* s.l. : Springer, 2013.

18. Shauer, Bryan. Multicore Processors – A Necessity. [En línea] https://web.archive.org/web/20111125035151/http://www.csa.com/discoveryguides/multicore/review.pdf.

19. Culler, David. *Parallel Computer Architecture: A Hardware/Software Approach.* 1999.

20. *Cluster Computing: A High-Performance Contender.* Baker, Mark, Buyya, Rajkumar y Hyde, Dan. 2000, Technical activities forum.

21. Knight, Will. IBM creates world's most powerful computer. *New Scientist.* june de 2007.

22. D'Amour, Michael R. *Standard Reconfigurable Computing.*

23. Boggan, Sha'Kia and Daniel M. Pressel. *GPUs: An Emerging Platform for General-Purpose Computation.* 2007.

24. *Exploring Multi-Grained Parallelism in ComputeIntensive DEVS Simulations.* Liu, Qi y Wainer, Gabriel. 2010, IEEE Workshope on principles of advanced and distributed simulation.

25. Daniels220. AmdahlsLaw. [En línea] Wikipedia. https://upload.wikimedia.org/wikipedia/commons/e/ea/AmdahlsLaw.svg.

26. *Validity of the single processor approach to achieving large scale computing capabilities.* Amdahl, Gene M. 1967, Spring Joint Computer Conference.

27. *Reevaluating Amdahl's law.* Gustafson, John L. 1988.

28. Gustaffson. [En línea] Wikipedia. https://en.wikipedia.org/wiki/File:Gustafson.png.

29. Singh, David Culler. *Parallel computer architecture.* San Francisco : Morgan Kaufmann, 1997.

30. Cburnett. FlynnsTaxonomy. [En línea] Wikipedia. https://en.wikipedia.org/wiki/Flynn%27s_taxonomy#/media/File:SIMD.svg.

31. Buyya, Rajkumar y Venugopal, Srikumar. *A Gentle Introduction to grid computing and technologies.* s.l. : Computer society of india, 2005.

32. Foster, Ian. *What is the Grid? A Three Point Checklist.* 2002.

33. Foster, Ian, Zhao, Yong y al, et. *Cloud Computing and Grid Computing 360-Degree Compared* .

34. e-sciencecity. What is grid computing? [En línea] https://www.e-sciencecity.org/EN/gridcafe/what-is-the-grid.html.

35. Coulouris, George, y otros. *DISTRIBUTED SYSTEMS. Concepts and designs.* s.l. : Addison Wesley, 2012. Capitulo 5.

36. Aho, Alfred V, y otros. *Compilers, principles, thniques and tools.* s.l. : Pearson Addison-Weasley, 2007.

37. Fox, Geoffrey, Williams, Roy y Messin, Paul. *Parallel Computing Works!* s.l. : Morgan Kaufman, 1994.

38. *Experiments in Separating Computational Algorithm.* Yehezkael, Rafael. 2000, Lecture notes in computer science of Springer, págs. 268-278.

39. Urbach, Eduard. Automatic parallelism and data dependency. [En línea] april de 2012. https://web.archive.org/web/20140714111836/http://blitzprog.org/posts/automatic-parallelism-and-data-dependency.

40. Campanioni, Simone, y otros. *The HELIX Project: Overview and Directions.* 2012.

41. Dept Computer Science Univarsity of Illinois. Charm++ Parallel programing framework. [En línea] http://charmplusplus.org/.

42. *CharmPy: A Python Parallel Programming Model.* Galvez, Juan J, Karthik, Senthil y Laxmikant, Kale. 2018, 2018 IEEE International Conference on Cluster Computing (CLUSTER).

43. University of California. BOINC. [En línea] University of California. https://boinc.berkeley.edu/.

44. Mengotti, Tiziano. GPU Protocol: Deifferences with centralized frmeworks. [En línea] http://gpu.sourceforge.net/gpu_p2p/node8.html.

45. Karau, Holden y Warren, Rachel. *High Performance Spark.* s.l. : O'reilly, 2017.

46. *Resilient Distributed Datasets: A Fault-Tolerant Abstraction for.* Zaharia, Matei y al, et. 2012.

47. OpenMP. Open MP. [En línea] https://www.openmp.org/.

48. Sanders, Jason y Kandrot, Edward. *Cuda by eExample.* s.l. : Addison-Wesley, 2011.

49. cuda-programming. [En línea] CUDA Programming. http://cuda-programming.blogspot.com/2012/12/thread-hierarchy-in-cuda-programming.html.

50. PLUTO - An automatic parallelizer and locality optimizer for affine loop nests. [En línea] http://pluto-compiler.sourceforge.net/.

51. *A practical automatic polyhedral parallelizer and locality optimizer.* Bondhungula, Uday, y otros. 2008, Proceedings of the 29th ACM SIGPLAN Conference on Programming Language , págs. 101-113.

52. Sandberg, Russel y al, et. *Design and Implementation of the Sun Network Filesystem.* 1985.

53. Smith, Christopher M. Linux NFS overview FAQ and HOWTO Documents. [En línea] http://nfs.sourceforge.net/.

54. Fonseca, Guillermo. NFS-Network File System. [En línea] https://guillermofonseca.wordpress.com/2011/03/31/nfs-network-file-system/.

55. Apache software foundation. HDFS Architecture Guide. [En línea] The Apache Software Foundations. https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html.

56. Reddy, Jayvardhan. HDFS Architecture in Depth. [En línea] Medium. https://medium.com/plumbersofdatascience/hdfs-architecture-in-depth-1edb822b95fa.

57. Python Software Foundation. General Python FAQ,. [En línea] Python Software Foundation. https://docs.python.org/2/faq/general.html#what-is-python.

58. —. History and License. [En línea] Python Software Foundation. https://docs.python.org/2/license.html.

59. Peters, Tim. The Zen of Python. [En línea] https://www.python.org/dev/peps/pep-0020/.

60. Lacchia, Michele. Welcome to radon documentation. [En línea] https://radon.readthedocs.io/en/latest/.

61. Pallets Team. Welcome to Flask. [En línea] http://flask.pocoo.org/docs/1.0/.

62. Beazley, David. PLY (Python Lex-Yacc). [En línea] https://www.dabeaz.com/ply/ply.html.

63. NON-SIMPLE PERIODIC ORBITS of the N-BODY PROBLEM. [En línea] http://adams.dm.unipi.it/~gronchi/nbody/nonsimple.

64. Torre de Hanoi. [En línea] Colegio Divina Pastora. http://www.colegiodivinapastora.com/blogs/matematicas/2012/10/26/torre-de-hanoi/.

65. Huttson, Keith. Understanding Intrusion Detection and Prevention Systems. [En línea] AccessAgility. https://www.accessagility.com/blog/understanding-intrusion-detection-and-prevention-systems.